SANDIA REPORT

SAND2011-8524
Unlimited Release
Printed November, 2011

An Introduction to LIME 1.0 and its Use in Coupling Codes for Multiphysics Simulations

Rod Schmidt, Noel Belcourt, Russell Hooper, Roger Pawlowski

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-8524
Unlimited Release
Printed November, 2011

An Introduction to LIME 1.0 and its Use in Coupling Codes for Multiphysics Simulations

Rod Schmidt, Noel Belcourt, Russell Hooper, and Roger Pawlowski
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

LIME is a small software package for creating multiphysics simulation codes. The name was formed as an acronym denoting "Lightweight Integrating Multiphysics Environment for coupling codes." LIME is intended to be especially useful when separate computer codes (which may be written in any standard computer language) already exist to solve different parts of a multiphysics problem. LIME provides the key high-level software (written in C++), a well defined approach (with example templates), and interface requirements to enable the assembly of multiple physics codes into a single coupled-multiphysics simulation code.

In this report we introduce important software design characteristics of LIME, describe key components of a typical multiphysics application that might be created using LIME, and provide basic examples of its use - including the customized software that must be written by a user. We also describe the types of modifications that may be needed to individual physics codes in order for them to be incorporated into a LIME-based multiphysics application.

Acknowledgment

The initial development of LIME was supported by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories. Additional development leading to version 1.0 of LIME was funded by the Consortium for Advanced Simulation of Light Water Reactors (CASL), a DOE Energy Innovation Hub for Modeling & Simulation of Nuclear Reactors.

Contents

Appendix

List of Figures

List of Tables

Chapter 1

Introduction

Many technical problems of great scientific or engineering interest involve complex systems with numerous, sometimes disparate, interconnected components that involve many different physical processes that interact in a variety of ways. Examples include the design, safety analysis and licensing of current and future nuclear reactors, development of renewable energy technologies, vulnerability analysis of water and power supplies, understanding of complex biological networks, and many others. In order to simulate these systems, relevant physical processes are described by models and equation sets that evolve the important state variables (e.g. temperature, density, velocity, neutron flux, etc.) in time over the spatial domain of relevance. When these processes interact with one another we say they are "coupled", and the equations must be solved in such a manner that changes in any state variable are properly reflected in all equation sets that are affected by that variable. If the effect of changes in variable "X" on the evolution of variable "Y" is small, then the coupling is said to be weak. Conversely, if the effect is large, then the coupling is said to be strong. Another important aspect of coupling is whether the effect is a linear effect, or a nonlinear effect. The most challenging problems to model accurately are those where the coupling is both strong and nonlinear.

LIME, an acronym for "Lightweight Integrating Multiphysics Environment for coupling codes," is a software package being developed to help create multiphysics simulation codes. LIME is intended to be especially useful when separate computer codes already exist to solve different parts of a multiphysics problem. Thus in this report we will use the word coupled in two related but different contexts; one with respect to computer codes, and one with respect to physics. In principle, a single physics code can be written to simulate any coupled multiphysics system, i.e. a single physics code is not restricted to a single-physics. Of interest to LIME is the ability for multiple such physics codes to be coupled in some fashion so as to accurately simulate a multiphysics system. In this case each physics code separately approximates the solution to one or more sets of physics in the overall coupled multiphysics system.

A sound understanding of the mathematical theory associated with multiphysics coupling is of fundamental importance for potential users of LIME. This topic is discussed in a separate companion report [7], where the basic theory associated with multiphysics coupling algorithms is described. Here we discuss important software design characteristics of LIME, the key components of a typical multiphysics application that might be created using

LIME, and provide basic examples of its use - including the customized software that must be written by a user. We also describe the types of modifications that may be needed to individual physics codes in order for them to be incorporated into a LIME-based multiphysics application.

The rest of the report is organized into three chapters, references, and an appendix. Chapter 2 provides an overview of LIME that

1. reviews LIME design objectives,
2. provides an introduction to code coupling algorithms, and
3. describes the key software components of LIME and how they are used to create a multiphysics application.

A small section discussing current limitations of LIME completes the chapter.

Chapter 3 provides a "quick start" that walks the user through the process of obtaining and building LIME and Trilinos on their own computer, and how to test that they run some example test problems.

Chapter 4 is a basic tutorial on many of the practical aspects of using LIME to create a multiphysics application. Three simple physics problems are introduced together with an associated set of small stand-alone computer codes. The chapter is organized into sections according to the major steps that might be followed when creating a new multiphysics application with LIME. Exactly how several different multiphysics applications can be created using the example physics codes is described in detail.

A fairly large appendix that contains all of the example code listings (Multiphysics Drivers, physics codes, Model Evaluators, etc.) is provided at the end of the report.

Chapter 2

Overview of LIME

2.1 Design Objectives

The main purpose of LIME is to provide a straight forward approach for efficiently creating a single-executable coupled-multiphysics simulation code when separate computer codes already exist to solve different parts of a multiphysics problem. It may also be useful when designing a multiphysics simulation code from scratch.

To enable the assembly of several codes into a multiphysics simulation capability, LIME provides the following:

- the key high-level software,
- a well defined approach (including example templates),
- interface requirements for participating physics codes, and
- direct access to advanced solver libraries such as Trilinos/NOX. [1][2]

In support of these objectives, the approach taken and software design is intended to:

- minimize the requirements barrier for an application to participate,
- enable both new and legacy applications to be combined with strong coupling (when needed) though non-linear solution methods (e.g. JFNK, fixed point),
- potentially preserve and leverage any specialized algorithms and/or functionality an application may provide.

LIME is specifically designed so that it is ***not limited*** to:

- codes written in one particular language,
- a particular numerical discretization approach (e.g. Finite Element), or
- physical models expressed as PDEs.

We describe LIME as "lightweight" because of the additional design objectives to

- keep the main software relatively small in size and complexity,

- require only a few standard and openly available libraries to build,
- be easily portable to a wide range of computing platforms, and
- minimize the constraints placed on the codes and models to be incorporated.

It is important to recognize that there are important tradeoffs closely linked to the LIME design objectives listed above. In order to create a new multiphysics application using LIME, some amount of customized software must be written and, depending on the original physics codes, some degree of refactoring may be required. In other words, LIME is not "plug and play" in the way that this phrase is normally interpreted. However, the effort required to create the customized software and to re-factor the physics codes of interest (which one might call "adapt and apply") may be significantly less than the alternative of creating an entirely new multiphysics code from scratch. An important aspect of this document is to describe what the customized software is, how it should be written, and the issues that must be addressed through the physics code refactoring step.

2.2 Introduction to Code Coupling Algorithms

When describing multiphysics simulation, we use the word "coupled" in two related but different contexts; one with respect to physics, and one with respect to computer codes. One computer code might be written to solve only one set of physics, but another might be designed to internally solve several coupled physical processes (i.e. a single physics code is not restricted to a single-physics). Of interest to LIME is the ability for two or more such physics codes (of either type) to be coupled so as to accurately simulate a multiphysics system. In this setting each physics code separately solves the equations for one or more sets of physics in the overall coupled multiphysics system.

The mathematical theory associated with multiphysics coupling in general, and multiphysics code coupling in particular, is of fundamental importance to LIME, and a separate report has been prepared as a more detailed primer on this subject [7]. Here we provide a brief introduction to the two basic code-coupling algorithms currently employed in LIME; fixed point coupling (called "Picard iteration" in [7]) and Jacobian-Free Newton-Krylov (JFNK) coupling [4]. We also briefly describe a "switching" solution mode (to be supported in later versions of LIME), the use of a "predictor" step when doing transient problems, and the concept of "nonlinear elimination," a mathematical technique by which the dependency of one physics code on the state variables from a different code can be implemented in a particularly useful way.

2.2.1 Fixed Point Coupling

Fixed point (or Picard) coupling is the simplest and perhaps most common approach to multiphysics coupling. Within LIME, fixed-point iterations are performed by sequentially

solving each physics code independently within a global iteration loop. Updated state variables are transferred between physics codes either at the end of each global iteration (here called a "Jacobi" method), or immediately after each physics code solve (here called a "Seidel" method). The Jacobi method is currently the default in LIME, but can be changed to the Seidel method by use of a user controlled input file as explained later. The sequence is repeated until a convergence measure is satisfied (e.g. the residual norms from all codes are below a prescribed tolerance), or a maximum number of iterations is performed.

Fixed point algorithms have several attractive characteristics;

1. they are simple to implement,
2. they are computationally cheap per iteration, and
3. they impose only minimal code requirements (essentially only the ability to do a stand-alone solve).

In many practical applications fixed point algorithms tend to be very robust. Robust operation means that an initial guess for the nonlinear solution may converge where the same initial guess may fail to converge for other coupling algorithms. (This property is also described by saying that it has a large radius of convergence.)

There are also important drawbacks to the the fixed point approach. First, there are stability requirements for the algorithm [5][6] that can be violated in some settings (i.e. the algorithm will not converge). Second, the convergence rate is relatively slow (mathematically q-linear in the norm [3]), and thus may take many iterations to converge to a stringent tolerance. This means that even though the cost of doing a single fixed-point iteration is small, the number of iterations required for convergence may be so large that performing a smaller number of more expensive iterations with JFNK may be more computationally efficient.

The ability to participate in fixed-point iterations is the lowest barrier to entry for an application to be coupled in LIME.

2.2.2 Jacobian-Free Newton Krylov (JFNK) Coupling

JFNK [4] is a powerful method for solving non-linear equations that is based on the classic "Newton's" method, but implemented with certain approximations and by leveraging a special class of iterative linear solver methods called "Krylov" solvers. To understand JFNK, one must be familiar with the basics of Newtons method, what a Jacobian is, and one important characteristic of the Krylov solution methods for linear systems.

Newtons method for finding the roots of a simple polynomial is introduced in most basic algebra classes. Using Newtons method in this setting, any root x of a polynomial $f(x) = a + bx + cx^2 + ... = 0$ can be found very quickly given two things; (1) a sufficiently

close guess x_0 for that root, and (2) the value of the derivative of $f(x)$, evaluated at x_0. The formula for Newtons method is expressed in terms of an iterative series

$$x_{n+1} = x_n - \frac{1}{df(x_n)/dx} f(x_n) \tag{2.1}$$

where x_{n+1} denotes the next estimate of x given the previous estimate x_n. If the value chosen for x_0 is sufficiently close to x then the iteration will converge to $x_n = x$ within numerical precision for large n.

The generalization of Newtons method to a coupled set of nonlinear equations is completely analogous to the application described above for finding the root of a polynomial. We begin by casting the solution to the equation set in terms of a residual vector $\bar{R}(\bar{x})$, where a valid solution $\bar{x}$ for the equation set is found when $\bar{R}(\bar{x}) = 0$. This is the "many equation" equivalent to finding the root of a single nonlinear polynomial equation. Using Newtons method in this setting, any solution vector $\bar{x}$ of the nonlinear equation set $\bar{R}(\bar{x}) = 0$ can be found given two things; (1) a sufficiently close guess for the solution vector $\bar{x}_0$, and (2) the inverse of the Jacobian matrix $\bar{J}^{-1}$, evaluated at $\bar{x}_0$.

The Jacobian matrix essentially describes the sensitivity of the residual vector to small changes in the state variables. The components of the Jacobian matrix $\bar{J}_{i,j}$ are defined as the derivative of each residual equation i with respect to each member of the state vector x_j, i.e.

$$\bar{J}_{i,j} = \frac{\partial \bar{R}_i}{\partial \bar{x}_j} \tag{2.2}$$

Newtons method can now be written in a more generalized form as

$$\bar{x}_{n+1} = \bar{x}_n - \bar{J}^{-1} \bar{R}(\bar{x}_n) \tag{2.3}$$

where $\bar{J}^{-1}$ denotes the inverse of the Jacobian matrix and $\bar{x}_{n+1}$ denotes the next estimate of $\bar{x}$ given the previous estimate $\bar{x}_n$. If the value chosen for $\bar{x}_0$ is sufficiently close to $\bar{x}$, then $\bar{x}_n = \bar{x}$ within numerical precision for large n.

Note that the size of the Jacobian matrix for a system of m equations with m unknowns is m^2. Thus for large values of m, direct solution methods for computing the inverse of the Jacobian matrix (equivalent to solving a linear system of equations) can become prohibitively expensive, and certain types of iterative methods have proven to be much more efficient. One important class of iterative solvers are referred to as Krylov solvers (examples include Conjugate Gradient (CG), GMRES, TFQMR, and so forth). In a Krylov linear solver the solution is built from a linear combination of matrix vector products, where the matrix is always the Jacobian matrix. The vector it is multiplied by changes each iteration and is a function of the particular method being employed.

Because even computing the Jacobian matrix itself (let alone computing its inverse) can be difficult or expensive, methods have been developed which apply the following first-order approximation for the action of the Jacobian matrix on an arbitrary vector $\bar{p}$.

$$\bar{J}p = \frac{\bar{R}(\bar{x} + \epsilon \bar{p}) - f(\bar{x})}{\epsilon}. \tag{2.4}$$

In a JFNK method, p is the Krylov vector mentioned above and ϵ is a perturbation parameter. In practice this requires a complete nonlinear residual evaluation of $\bar{R}(\bar{x} + \epsilon p)$ during each Krylov iteration, but avoids the expensive cost of forming the Jacobian matrix itself (thus the name "Jacobian-Free").

In summary, a Jacobian-Free Newton Krylov (JFNK) method derives its name from the fact that it has the following characteristics.

- The solution to the non-linear problem is found using Newtons method.
- A Krylov method is used to solve the linear system of equations needed to compute $\bar{J}^{-1}\bar{R}(\bar{x}_n)$ in Eq. 2.3.
- The approximation of Eq. 2.4 is used by the Krylov solver for all matrix vector products so that the Jacobian matrix never has to be formed.

In practice, a mathematical operation called matrix "preconditioning", is often needed to accelerate the Krylov solve. This procedure typically uses a matrix that approximates the true Jacobian, and a variety of different types of preconditioning methods are described in the literature. This aspect of using the JFNK method is not discussed here, but a good introduction is provided by [4].

In the context of LIME, we simply note that if a physics code can return a residual vector $\bar{R}(\bar{x})$ for any approximate solution vector $\bar{x}$, then the JFNK solution method can be applied. As a point of clarification, the values of the scalar ϵ and the vector $\bar{p}$ shown in Eq. 2.4 are computed as part of the solution methodology implemented within the nonlinear solver (NOX), not by the physics code. When using JFNK in LIME, the Problem Manager registers itself with NOX as the callback interface for residual fills and aggregates data from application code Model Evaluators (e.g. residuals) to send to NOX as well as segregates data (e.g. solution state) from NOX to send to respective Model Evaluators. Value-added enhancements come from the ability of the Problem Manager to aggregate additional data, e.g. a physics code's Jacobian matrix, or additional functionality, e.g. the action of a physics code's specialized preconditioner, into the JFNK algorithm.

JFNK has two important advantages over the fixed-point methods;

1. The iteration converges quadratically to the solution when the initial guess is sufficiently close, and
2. The method is robust for almost all problem types (i.e. no stability issues).

Drawbacks to the JFNK approach are primarily the following.

1. The requirement that a physics code must be able to compute and return a residual vector $\bar{R}(\bar{x}_n)$ given any arbitrary state vector $\bar{x}_n$ when requested by the Problem Manager.
2. The cost of performing a linear solve each Krylov step.

3. The requirement that the initial guess be within the "radius of convergence." Globalized Newton methods [3] such as line search and trust region methods provided by NOX can remove the restriction of a sufficiently close initial guess, however this typically increases the cost of each nonlinear iteration.
4. That good performance may require preconditioning operations.

2.2.3 Switching between fixed-point and JFNK

In some settings, it may be useful to begin a solution strategy with a fixed-point algorithm and then switch to the JFNK method after several iterations. This hybrid method can be useful when the initial guess for the solution vector is outside of the radius of convergence for the JFNK method. As mentioned above, in some situations the fixed point method will converge, albeit slowly, for initial guesses that are quite poor. Thus a switching strategy that uses fixed point to improve the estimate, and then switches to JFNK to more quickly drive the estimated solution to a fully converged state, could be very useful for some problems.

Although this capability has been tested within LIME [8], it is not supported in the current release. This functionality, together with a wide range of additional options available in the Trilinos/NOX solver library (see [1][2]), is being explored for future implementation. However, how best to manage the complexity of these options within a user-friendly interface has not yet been determined.

2.2.4 Using a predictor step for transient problems

For transient problems, LIME offers the option for codes to provide a "predicted" solution based on previous time-step values of all state variables prior to entering the non-linear solution algorithm (see Section 2.3.2.3, Table 2.1). This is equivalent to doing an explicit time integration step to provide an improved initial estimate to whatever solution method is being employed. If JFNK is being used, providing a predicted solution is essentially doing one iteration of the fixed point method (using the "Jacobi" option described above) prior to beginning the JFNK algorithm. In essence, this is a limited form of the switching approach just described for transient problems.

2.2.5 Non-linear Elimination

In LIME, nonlinear elimination refers to an approach by which one or more of the coupled codes are eliminated from the global solve process by enforcing the eliminated codes to be in a constantly converged state. A discussion of the mathematical basis is given in [7].

From a practical standpoint, it simply means the following. Consider two codes, A and B, which solve for state variables $\bar{x}_A$ and $\bar{x}_B$ respectively, but where the solution of the

equations in physics code A also depends in some manner on the state variables in physics code B, and vice versa (i.e. the equation sets are coupled). One strategy for performing a global coupled solve is to treat physics A as primary, i.e. as if it is the only physics being treated. We then arrange the solution algorithm to compute the $\bar{x}_B$-dependent quantities whenever they are needed by performing a complete solve to convergence of the physics B equation set and then computing the quantities required. This is analogous to calling a subroutine that, given certain inputs, provides back the desired outputs. We will call this type of model a "response-only" model, and will refer to a physics code that is "eliminated" from the problem set by using non-linear elimination as an "elimination module".

One important feature of the Problem Manager in LIME is the ability to incorporate into the JFNK coupling algorithm a physics code that only provides a response (i.e. output) given appropriate input. The Problem Manager incorporates these response-only Model Evaluators into the JFNK algorithm using nonlinear elimination. Every time NOX requests a computation of the residual vector for what it views as a single-code nonlinear problem, the Problem Manager converts the state vector provided by NOX into the appropriate inputs (via data transfer Model Evaluators) to the response-only Model Evaluators, triggers their solve method and converts the responses into the forms needed by dependent physics codes before calling the residual computations for the Model Evaluators that can provide residuals. This approach captures the equation-to-variable sensitivities of all physics codes being coupled in a manner consistent with Newtons method.

The major advantage of nonlinear elimination is that each code is free to use the best algorithm for solving its partcular physics. The drawback is that every time a residual or Jacobian evaluation of the outer physics is required, all elimiated physics must perform a complete nonlinear solve. If the elimiated phyiscs are expensive, the algorithm can be very costly compared to JFNK coupling.

2.3 Components of LIME

Figure 2.1 illustrates the key components of LIME in the context of a generic multiphysics application that has been created using LIME from three individual physics codes.

At the highest level (level 3 in the illustration) LIME currently consists of two parts, (1) a "Multiphysics Driver" and (2) a "Problem Manager" which is linked to the Trilinos/NOX non-linear solver library. Each of these are written in C++. The Multiphysics Driver is very small (typically less than 100 lines) and must be customized by a LIME user to reflect the particular needs of the application being created. In contrast, the Problem Manager consists of several modestly sized C++ files (of order several thousand lines of code), but is not modified by a user. Optional input files can be used to control many aspects of how the Problem Manager will operate.

At the middle level are an arbitrary number of LIME "Model Evaluators," one for each physics code being coupled. Model Evaluators must be written in C++ by the LIME user

and usually serve to transform (or wrap) a previously stand-alone physics code into a call-back subroutine that can communicate with the Problem Manager. They can also be used to perform special transfer operations that may be needed to pass information between physics codes. A Model Evaluator may range in size from only a few lines long to several hundred lines long, depending on the situation.

At the lowest level of the hierarchy are the physics codes themselves, which only communicate directly to their respective Model Evaluator. Each of these may have associated with them user-defined input files and perhaps certain output files that will be generated (not shown). Physics codes can be written in any standard computer language. Depending on the original physics codes, some degree of refactoring may have been required for these codes to properly interface with its respective Model Evaluator.

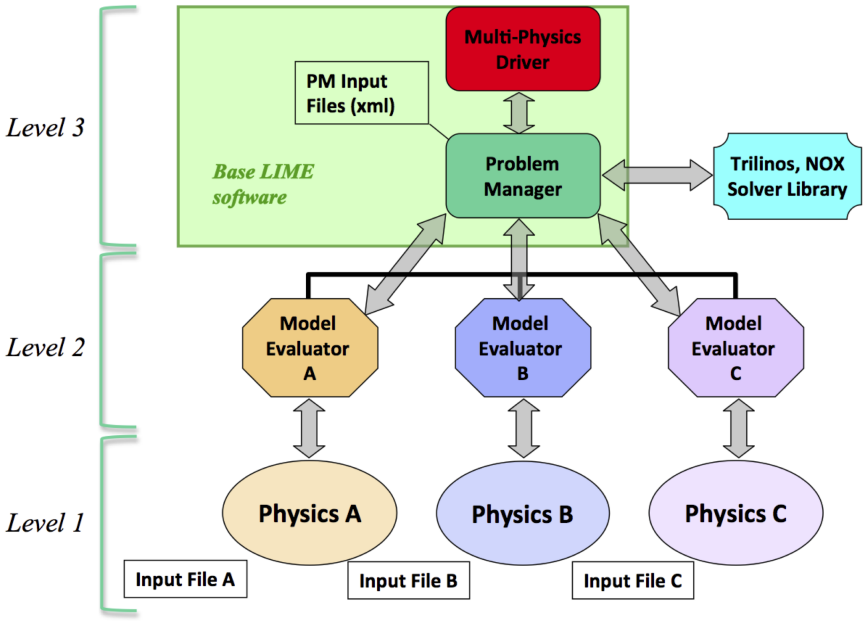


Figure 2.1. Key components of a simple generic application created using LIME

The sub-sections that follow provide additional details about each of these components.

2.3.1 The Multiphysics Driver

The Multiphysics Driver is not a formal part of LIME but instead represents the highest-level software component responsible for instantiating, configuring and invoking the LIME Problem Manager. It is basically the main program that is specialized to whatever applications and associated Model Evaluators are being coupled within a multiphysics simulation. The Multiphysics Driver performs the following three administrative functions:

1. Do all set-up tasks for the problem that is to be run, including creating an instance of the Problem Manager specific to the multiphysics application being generated. The set-up phase includes the following specific tasks;

    - Create an "Epetra" communicator object. LIME uses the Trilinos Epetra package for its concrete implementation of vector and linear operator services. The "Epetra Comm" virtual class is an interface that encapsulates the general information and services needed for the other Epetra classes to run on a serial or parallel computer. If running in parallel, the communicator object will be based on an MPI communicator and the number of processors requested. If serial, this will be an instance of an "Epetra_SerialComm" object.
    - Create a Problem Manager object having the designated communicator.
    - Register the Model Evaluators for each physics package being coupled with the Problem Manager.
    - Create and setup the LIME data transfer operators that will be needed.
    - Register the transfer operators with the Problem Manager.

2. Call the Problem Manager to solve the problem
3. Gracefully end the simulation

The Multiphysics Driver is a customized piece of software that must be written by the LIME user for the particular multiphysics problem being created. However, the length of this file is very small, and the format follows a standard pattern. Exactly how to write a LIME Multiphysics Driver is explained later in Chapter 4 by use of several examples, each of which can be adapted for other use cases.

2.3.2 The Problem Manager

The Problem Manager can be thought of as the central orchestrating software component that controls the multiphysics simulation being performed. It is designed to encapsulate the various Model Evaluators being coupled, to drive various types of fixed-point coupling iterations or, if using a JFNK type solver, to present a single-problem view to the Trilinos solver framework. We note that in LIME we have adopted a design for the Problem Manager that is an extension to a prototype multiphysics coupling capability in the NOX nonlinear solver package in Trilinos [1]. The Problem Manager API has been expanded in order to incorporate more functionality embodied in physics codes as well as to lower the barrier to entry for physics codes to participate in coupling algorithms. In addition, the Problem Manager in LIME can be configured from XML input files in a manner consistent with how other solver packages in Trilinos are configured.

In this section we describe various aspects of the Problem Manager with text and tables. The examples described in Chapter 4 should also be reviewed in order to better understand this material from an actual implementation standpoint.

2.3.2.1 Overview

The Problem Manager registers physics codes and associated data transfers wrapped as Model Evaluators (by requiring that they inherit an EpetraExt ModelEvaluator base class). It can then query each Model Evaluator to ascertain what input arguments it can receive and what output arguments it can provide given the input arguments. Based on the results of the queries, various coupling algorithms are possible, and the Problem Manager checks to make sure its configuration settings from an input file are compatible with the functionality available from the registered Model Evaluators. For example, the Model Evaluator for Physics Code A can indicate that it supports a state, $\bar{x}_A$, and with this can compute and provide a residual $\bar{R}_A(\bar{x})$. This would permit it to participate in various Newton-based coupling algorithms. Applications unable to provide residuals can only participate in a coupling algorithm via nonlinear elimination or fixed-point. If a physics code can provide additional callback support such as its own Jacobian, the structure of its Jacobian or its own specialized solution method, these can be incorporated into the Jacobian or a preconditioner or both for the complete coupled problem.

At present the Problem Manager supports time stepping of the various applications by requiring applications to supply a candidate time step size and then negotiating a most conservative value to be used by all. Extension to more elaborate operator splitting methods is possible but is not currently supported. Hooks are also provided for pre- and post-time step functionality to enable such functionality as cycling time plane values, I/O, computing derived quantities, computing a predictor, check-pointing a solution, etc.

Because each application is wrapped as a Model Evaluator, the Problem Manager can apply a variety of coupling solver algorithms, ranging from fixed-point approaches to various implementations of the JFNK method (see Section 2.2 above). In addition it is also possible to mix coupling solver algorithms so that some parts of the overall coupled system are treated using fixed-point while others are solved using JFNK. This may be desirable in cases where the dependence among some applications is either weak or inherently one-way. (This mode is possible but has not been tested as of yet.) In all cases, transfers of data among the applications must be performed in a manner consistent with each application's requirements, e.g. temperature from A to B and heat flux from B to A. These transfers can range from very simple copies of data to very elaborate mesh interpolations of derived quantities. The Problem Manager accommodates data transfers of general complexity by simply treating them as additional Model Evaluators. Any specialized solution techniques that are needed to perform the transfer can be implemented by the user.

2.3.2.2 Key Tasks

The key tasks of the Problem Manager can be outlined as follows.

1. If required, create the global state vector X representing the degrees of freedom for the single-problem view of the coupled problem and define mappings between X and the input arguments to each Model Evaluator representing either a physics code or a data transfer. Some configurations for fixed point coupling do not require this task.

2. Configure and initialize the fixed-point and/or JFNK coupling solvers.
3. Solve the global equation set. If integrating over time, including the following for each time step:
   - Perform time-step control: Negotiate/calculate based on all the physics.
   - Optionally request a"predicted" solution state for time step n+1 from each physics code through its Model Evaluator.
   - Obtain a converged solution for time step n+1 using a non-linear solution strategy defined by the user input. Current options include simple fixed point iteration and JFNK using the Trilinos/NOX nonlinear solver library. This may optionally include subtasks such as requesting residuals from physics codes or implementing physics-based preconditioning.
   - Update state vectors and time
   - Perform output (each physics code)
4. Return control to Multiphysics Driver

2.3.2.3 User control through input files

Nonlinear solver options and convergence/stopping criteria for use by the Problem Manager can be configured by a user through the following three XML-based input files.

- Problem_Manager_setup.xml: This input file is used to specify the global solution method that will be used and whether or not the solution will begin with a predictor step. The parameters set in this file are listed in Table 2.1.
- Problem_Manager_setup_fp.xml: This input file is used with the fixed point solver strategy. It specifies the type of fixed point algorithm, and sets the maximum number of iterations that will be attempted to achieve convergence to an optionally specified tolerance. When residuals are not supported, convergence is determined by calls to each individual physics code. The parameters set in this file are listed in Table 2.2.
- Problem_Manager_setup_jfnk.xml: This input file is used with the JFNK solver strategy. It specifies if preconditioning will be used, and sets the maximum number of iterations that will be attempted to achieve convergence within an optionally specified tolerance. The parameters set in this file are listed in Table 2.3.

The presence of these Problem Manager input files is optional because default values are assigned for all parameters. Each provides the user with the ability to configure the behavior of the Problem Manager in different ways. Specific examples of each of these input files and their use in solving various example problems is given later in Section 4.3.

In the current version of LIME the number of parameters exposed through these input files is relatively small. A larger set of input parameters that exposes other functionality, including the many options available in the Trilinos/NOX solver library, is being explored for future implementation. However, how best to manage the complexity of these options within a user-friendly interface has not yet been determined and thus is not described here.

Parameter name	type	Valid values (default blue)	Notes
Solver Strategy	string	**FIXED-POINT** JFNK	JFNK solver strategy option requires that residuals be supported by the model evaluators.
Use Predictor	bool	**false** true	Implies a time dependent problem. Can be equivalent to a single "jacobi" iteration prior to doing the main solve. But also allows individual applications to leverage their own internal predictor functionality.
Max Time Steps	int	Any pos. integer **largest int supported by compiler**	Limits the max number of time steps to take in a transient simulation. The actual number taken will be the minimum of this value and the values specified in all Model Evaluators that are transient (using get_max_steps).

Table 2.1. Parameters set in the file Problem_Manager_setup.xml

Parameter name	type	Valid values (default blue)	Notes
Solve Mode	string	**Jacobi** Seidel	
Maximum Iterations	int	Any pos. integer **1**	Max. number of global iterations taken by the fixed-point algorithm to achieve convergence.
Absolute Tolerance	double	Any pos. number **0.0**	Ignored (with warning) if residuals are not supported. L2 norm of the global residual vector is compared against this number to determine convergence.

Note: By setting the default tol. to zero, the code will always take the max number of iterations, which by default is only 1.

Table 2.2. Parameters set in the file Problem_Manager_setup_fp.xml

Parameter name	type	Valid values (default blue)	Notes
Preconditioner Operator	string	**None** Finite-Difference Physics-Based	"Physics-Based" option requires that special routines be supported by the model evaluator.
Maximum Iterations	int	Any pos. integer **1**	Max. number of global iterations taken by the JFNK algorithm to achieve convergence.
Absolute Tolerance	double	Any pos. number **0.0**	L2 norm of the global residual vector is compared against this number to determine convergence

Table 2.3. Parameters set in the file Problem_Manager_setup_jfnk.xml

2.3.3 Model Evaluators

In LIME, a Model Evaluator is a customized piece of C++ software that serves to transform (or wrap) an individual physics code into a call-back subroutine that can communicate with the Problem Manager. This capability forms the basis for it to participate in multi-physics coupling and a Model Evaluator must be written for each physics code being coupled with LIME. In LIME 1.0 the supported Model Evaluator capability is contained in a C++ Model_Evaluator class that includes extensions to the Trilinos EpetraExt::ModelEvaluator base class.

Model Evaluators enable data to be communicated to/from the Problem Manager, other Model Evaluators, and to its associated physics code. Each Model Evaluator is typically unique, and is based on

1. the characteristics of the multiphysics problem being solved,
2. the solution methods chosen by the user,
3. the particular physics code it controls, and
4. the set of interacting physics codes whose respective state variables are needed by the model equations being solved in the physics code.

Each Model Evaluator oversees that part of the overall state vector associated with its physics code. It also must have access to all other state vector pieces that appear directly, or indirectly (e.g. through property variation effects) in its equation set or models.

2.3.3.1 Basic implementation requirements

Each Model Evaluator inherits the LIME "Problem_Manager" base class and must implement the supported Problem_Manager/Model_Evaluator interfaces in terms of calls to the underlying physics package. Each Model Evaluator will normally have at least the following three parts (but may include many other parts depending on the problem requirements).

- a C++ constructor, where any physics code setup (e.g. read input files) is implemented.
- an implementation of either a "solve_standalone" or a "perform_elimination" interface routine where the physics code specific solve routine is called.
- a C++ destructor, where any final computations or desired output is implemented.

As a simple example, Figure 2.2 shows the source code for a bare-bones generic stand-alone Model Evaluator, here called PHYA_ModelEval.cpp (note that the separate header file is not shown). By "stand-alone" we mean that only a single physics code has been wrapped by LIME and therefore no code coupling is defined. In this case the physics code it controls is assumed written in Fortran 90 and has three high-level subroutines; setup_phyA, solve_phyA, and finish_phyA. As can be seen in this listing, the Model Evaluator constructor requires one

```
// ***********************************************************************************************
// PHYA_ModelEval.cpp
// This is a dummy bare-bones LIME model evaluator for a standalone wrap of a dummy
// fortran physics code called phyA.f90
// ***********************************************************************************************
#include "PHYA_ModelEval.hpp"
#include "LIME_Problem_Manager.hpp"

// This is how fortran routines are referenced within C++. The "name mangling"
// convention used here is specific to the intel fortran compilor.

 extern "C" {
   void setup_phyA_();     // the phyA code's setup  routine
   void solve_phyA_(int*); // the phyA code's solve  routine
   void finish_phyA_();    // the phyA code's finish routine
 }

//------------PHYA_ModelEval C++ constructor -----------------------------------------

PHYA_ModelEval::PHYA_ModelEval(LIME::Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name)
{
  setup_phyA_();           // This calls the phyA code's setup routine
}

//------------PHYA_ModelEval solve_standalone ----------------------------------------

void PHYA_ModelEval::solve_standalone()
{
    solve_phyA_();   // This calls the phyA code's solve routine
}


//------------PHYA_ModelEval C++ destructor ------------------------------------------

PHYA_ModelEval::~PHYA_ModelEval()
{
   finish_phyA_();         // This calls the phyA code's finish routine
}

//
// Other Problem_Manager interfaces that need to be implemented should follow next
//
```

Figure 2.2. Listing of a generic bare-bones stand-alone Model Evaluator file for LIME

argument, the LIME Problem_Manager, that owns and orchestrates this model evaluator's participation in a multi-physics coupling. An optional descriptive name can be supplied if desired, and this name will be used to output error messages originating from this Model Evaluator.

2.3.3.2 Model Evaluator Interfaces

There are a range of different capabilities currently supported in the LIME Model Evaluator class. However, most physics will only implement a few of the available features and the remaining features have reasonable defaults so that users can safely ignore all Model Evaluator interfaces except for those features that their physics codes actually support. In addition, many of the features in the Model Evaluator class are disabled by default. Features are

typically enabled in the Model Evaluator header file by implementing the supports_feature() method and returning boolean true, to indicate that a physics code is able to support this capability. An example is the supports_standalone_solve() feature seen in Fig. 2.2. If a physics code is able to perform a standalone solve, then the model evaluator class must notify the Problem Manager of this ability by returning true from this function.

In order to use any capability, an interface routine that meets required specifications must be included in the source file of the Model Evaluator where the capability is needed. Our first example is seen in Fig. 2.2, where a "solve_standalone" interface is implemented.

LIME Model Evaluator capabilities can be categorized into the following groups.

- time integration and stepping,
- residual support,
- preconditioning,
- predictors and variable scaling,
- convergence notification and testing, and
- parameter sensitivity response functions

This introductory report does not provide a description of all of these capabilities and their associated interfaces. Instead, we focus on a subset of some of the most important, and illustrate their use through specific examples. Table 2.4 lists the important Model Evaluator interface routines that are introduced here. Their use is illustrated in the examples provided in Chapter 4.

Interface Routine Name	Type	Description
supports_standalone_solve	bool	Return **true** or **false** to indicate whether the physics code supports a standalone solve capability.
supports_residual	bool	Return **true** or **false** to indicate whether the physics code can compute and return a residual through residual fill callbacks (see evalModel interface below).
is_transient	bool	Return **true** or **false** to indicate whether the physics code is solving time dependent equations.
solve_standalone	void	Perform a stand-alone solve of the physics code's equations. This will yield updated state variable values.
perform_elimination	bool	Perform a stand-alone solve of the physics codes equation set given current state variable values from another physics code. Return **true** or **false** to indicate successful completion.
get_time_step	double	Return the desired time step from the physics code.
get_max_time	double	Return the maximum time to temporally integrate to before completion.
get_current_time	double	Return the current time according to the physics code.
get_max_steps	unsigned int	Return the maximum number of time steps to take before stopping the time integration procedure.
update_time	void	Update the time in a physics code (typically after a converged solution to the coupled equation sets has been found).
set_time_step	void	Set the time step in a physics code solving transient equations.
is_converged	bool	Return **true** or **false** to indicate whether the physics code convergence criteria has been met.
initializeSolution	void	Initialize the state variables of the physics code.
set_x_state	void	Copy a specified incoming vector into the physics code's state vector.
get_x_state	Epetra_Vector	Copy current state vector from the physics code into a LIME array of type ``Teuchos::RCP<Epetra_Vector>''
get_x_init	Epetra_Vector	Copy current state vector from the physics code into a LIME array of type ``Teuchos::RCP<Epetra_Vector>'' to initialize the LIME global state vector.
createInArgs	See description ->	Sets up and returns an EpetraExt:ModelEvaluator data type called ``inArgs'' that describes input arguments. For example, we typically need to indicate that we can do calculations with incoming state x.
createOutArgs	See description ->	Sets up and returns an EpetraExt:ModelEvaluator data type called ``outArgs'' that describes output arguments. For example, we typically need to indicate that we can compute a residual vector.
evalModel	void	If a physics code supports computing a residual, then the evalModel routine is the core callback mechanism used by Problem_Manager to obtain desired outArgs given quantities passed via inArgs. For example, outArgs may contain valid (i.e. non-NULL) residual and Jacobian pointers signaling a request to fill them using the state passed in through inArgs.

Table 2.4. Important LIME Model Evaluator interface routines.

2.3.3.3 Required Interfaces

Depending on the application and solution algorithms, different interfaces may need to be implemented for a particular multiphysics application. For example, if a transient problem is being solved, then interfaces for specifying or computing time-related quantities must be properly implemented. At present, these would include "get_time_step", "get_max_time", "get_current_time", "get_max_steps", and "update_time".

Tables 2.5 to 2.7 catalog the required Model Evaluator interface routines that are associated with the different problem types and input parameters described previously in Section 2.3.2.

Examples designed to provide additional details concerning the use and design of Model Evaluators in LIME are provided in Chapter 4.

XML input file option	Set in	**Requirements**
Solver Strategy = JFNK	Problem_Manager_setup.xml	At least one ME must support residuals. Thus the following must appear in the me.hpp file **virtual bool supports_residual() const { return true; }** ***all interfaces listed Table 2.7 below*** **set_x_state** Note: For JFNK, a ME must either support residual or support elimination. If it can do both, it will default to using the residual.
Solver Strategy = FIXED-POINT	Problem_Manager_setup.xml	At least one ME must support standalone_solve, which means the following must appear in the me.hpp file: virtual bool supports_standalone_solve() const { return true; }

Table 2.5. Implied requirements for XML input file options.

Problem configuration	Set in	**Required** or optional Model Evaluator Interface Routines
virtual bool supports_standalone_solve() const { return true; }	me.hpp	**solve_standalone()** initializeSolution()
virtual bool supports_standalone_solve() const { return false; } and does support Residual but not JFNK	me.hpp	perform_elimination()
virtual bool supports_standalone_solve() const { return false; } and doesn't support Residual	me.hpp	**perform_elimination()**
virtual bool supports_standalone_solve() const { return false; } and Solver Strategy = FIXED-POINT	me.hpp	**perform_elimination()**
virtual bool is_transient() const { return true; }	me.hpp	**get_time_step()** **get_max_time()** **get_current_time()** **get_max_steps()** **update_time()**
virtual bool supports_residual() const { return true; }	me.hpp	**See Table 2.7 below**
virtual bool supports_residual() const { return false; }	me.hpp	**is_converged()**

Table 2.6. Model Evaluator interface requirements for different problem configurations.

Additional lines me.hpp	Interfaces to implement in me.cpp
virtual EpetraExt::ModelEvaluator::InArgs createInArgs() const;	createInArgs() const
virtual EpetraExt::ModelEvaluator::OutArgs createOutArgs() const;	createOutArgs() const
virtual void evalModel(const InArgs&, const OutArgs&) const;	evalModel(. . .) const
virtual Teuchos::RCP<const Epetra_Vector> get_x_init() const;	get_x_init() const
virtual Teuchos::RCP<Epetra_Vector> get_x_state();	get_x_state()
virtual Teuchos::RCP<Epetra_Vector> get_x_state() const;	get_x_state() const
-	initializeSolution()
virtual Teuchos::RCP<const Epetra_Map> get_x_map() const { return epetra_map_; }	-
virtual Teuchos::RCP<const Epetra_Map> get_f_map() const { return epetra_map_; }	-

Table 2.7. Model Evaluator interface requirements for supporting residuals.

2.3.3.4 Data Transfer Operators

Whenever two codes are coupled within LIME, a Data Transfer Operator must be created to define the transfer of data between the codes. If the coupling is two way (i.e. code A depends on code B as well as code B depends on code A), then two distinct Data Transfer Operators must be created.

Each implementation of a Data Transfer Operator requires two things, (1) a C++ constructor and (2) a member function called "perform_data_transfer." These can be placed within the Model Evaluator of either one of the two codes between which data is being transferred (or elsewhere if desired). The actual transfer is defined in the "perform_data_transfer" member function, and can occur in many different ways. Often it is done very simply, such as a standard copy function call. For example, lets assume that we need to transfer three temperatures from a temperature array in "codeA" to one of the same name in "codeB". We might add the following lines to the Model Evaluator for codeA:

```
//-------Constructor for codeA to codeB Data Transfer Operator --------------------------------
A_2_B::A_2_B(int from_id, int to_id, LIME::Problem_Manager & pm)
              : LIME::Data_Transfer_Operator(from_id, to_id), problem_manager_(pm)
{
}


//------- Simple implementation to define the actual data transfer from codeA to codeB ---------
bool A_2_B::perform_data_transfer() const
{
   Teuchos::RCP<LIME::problem_manager_api> & A_pm =  problem_manager_.get_problem(source_id());
   Teuchos::RCP<LIME::problem_manager_api> & B_pm = problem_manager_.get_problem(target_id());

   CODEA_ModelEval* codeA_me =  dynamic_cast<CODEA_ModelEval*>(&(*codeA_pm));
   CODEB_ModelEval* codeB_me =  dynamic_cast<CODEB_ModelEval*>(&(*codeB_pm));

  std::copy(codeA_me->t, codeA_me->t+3, codeB_me->t);
  return false;
}
```

The Multiphysics Driver must also be properly written for the data transfer to work within a LIME application (see 2.3.1). In the Multiphysics Driver we must do the following two things.

1. Create an instance of each Data Transfer Operator that is to be used. This action references the specific Data Transfer Operator name (implemented in a Model Evaluator) and defines the two physics problem IDs between which data is to be transferred.
2. Register (or "add") each transfer operator with the Problem Manager. Currently, a Data Transfer Operator can be added as one of three types, called "transfer", "pre_elimination_transfer" or "post_elimination_transfer". The pre and post elimination type designation specifies when the transfer should occur relative to a solve if the coupling is associated with a non-linear elimination. The type is ignored if the coupling is not associated with non-linear elimination.

Several examples are discussed in Chapter 4 to further illustrate how Data Transfer Operators are set up in LIME.

2.3.4 Physics codes

In principle, few restrictions are associated with the kinds of models or physics codes that can be coupled using LIME. We have noted earlier that LIME is specifically designed so that it is not limited to codes written in one particular language, a particular numerical discretization approach (e.g. Finite Element), or physical models expressed as PDEs. However, in order to interface with LIME, some modifications are typically required. These include the following.

1. Console IO must be redirected (no pause statements or read/write to standard streams (cin, cout, cerr, clog) are allowed).

2. The physics code must be written so it can be compiled as a library. This enables LIME to link to it (i.e. no "main").

3. The physics code must be organized into several key parts that can be called independently, and which correspond to the interfaces that its Model Evaluator must implement. At the very least LIME requires the ability to call routines that perform the following.

   - Read any required input and initialize the code
   - Solve the equations
   - Gracefully finish and perform any final output

4. Depending on the Problem_Manager/Model_Evaluator interface requirements for the problem type and solution method being used, additional routines may need to be available or added. For example, you might need routines to

   - Pass control variables (e.g. time step, max time, etc.)
   - Compute and pass data for coupling to other physics codes
   - Compute residuals
   - Perform preconditioning (e.g., physics based preconditioning) for JFNK
   - If the physics code supports MPI, then a function to coordinate MPI with the Multi Physics Driver must be written

One of the most important things to note is that global data (e.g. F77 common blocks, F90 modules, C extern and static) must be treated with care to avoid problems with subtle errors for non-mangled names (e.g. link data symbol instead of identically named code symbol), and because global or static data can inhibit threading. For large complex legacy codes written in earlier versions of Fortran, issues such as these may be the most significant issues that must be addressed in order to couple using LIME.

The process of refactoring is illustrated by a set of specific examples in Chapter 4.

2.4 Some Current Limitations of LIME

LIME remains a work in progress with a variety of improvements that could be made. Here we note several current limitations of LIME that a user/developer should be aware of.

2.4.1 Concerning Time-integration

Currently, LIME can only support simple two-step time integrations schemes such as the Crank-Nicholson or Euler methods. An important improvement being looked at is how best to support codes that leverage higher-order multi-step methods. In addition, LIME does not require that the time integration methodologies used by the different physics codes are consistent. Thus, if a user/developer is not careful, significant time integration errors could be introduced by not carefully addressing this important issue.

LIME uses a very simple method for determining the limiting time-step. Currently, each code can be queried to provide a minimum time step value to take for the next step. The actual step taken is chosen as the minimum value returned.

2.4.2 Concerning Running Codes in Parallel

LIME allows MPI based parallelism in the physics codes that can support this. However, LIME does not currently address parallel work flow management at the Problem Manager level. Rather, each MPI-based code must handle its own parallel decomposition and computation issues independently.

2.4.3 Concerning Multi-Threading Support

LIME is currently based on Epetra data structures in Trilinos. This comes with all the advantages and disadvantages represented by Epetra and Epetra-based solver packages in Trilinos. One disadvantage is the absence of thread-safe functionality in various parts of Trilinos. This deficiency is being addressed by the Trilinos developer community, and LIME will aim to take advantage of progress in this area as it is made.

Chapter 3

LIME QuickStart

The purpose of this chapter is to provide a very short concrete exposure to LIME. This will allow the reader/user to follow along with later descriptions of examples, interfaces, etc. in a hands-on manner.

3.1 Quick Start

The following sections will walk the user through the steps to get LIME built and running the examples.

3.1.1 Obtain Trilinos & LIME

The first step is to obtain the Trilinos and LIME source code. Official releases of Trilinos are available as downloadable tarballs from: http://trilinos.sandia.gov. LIME requires Trilinos version 10.X or later.

Currently, the LIME source code is not openly accessible. Please contact the lime development team to request obtaining the source code (limeext-developers@software.sandia.gov). Once LIME is obtained, place the LIME source code inside the top level Trilinos directory.

3.1.2 Build Trilinos & LIME

The next step is to build the Trilinos and LIME libraries and optionally the LIME examples. Trilinos and LIME use Cmake for their builds. A "best practices" approach for building both is to create a build subdirectory along with a cmake build script suited to whatever build types are desired, e.g. optimized, debug, other options enabled or disabled. A simple debug build of Trilinos and LIME can be performed as follows:

```
mkdir LIME_DEBUG_BUILD
cd LIME_DEBUG_BUILD
```

A sample cmake configure file that can be copied into the build subdirectory and made executable is shown in Figure 3.1.2. The line specifying examples be built could optionally be

```
#!/bin/bash

TRILINOS_PATH=~/Trilinos # top-level Trilinos directory

cmake \
        -D CMAKE_INSTALL_PREFIX="/usr/local" \
        -D Trilinos_EXTRA_REPOSITORIES="LIMEExt" \
        -D Trilinos_ENABLE_LIME:BOOL=ON \
        -D Trilinos_ENABLE_DEBUG=ON \
        -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF \
        -D LIME_ENABLE_EXAMPLES:BOOL=ON \
        ${TRILINOS_PATH}
```

Figure 3.1. Sample cmake configure script. Note: Ensure that the last character in all but the last line is the backslash '\'.

excluded or the argument changed to OFF in order to build only LIME libraries. Assuming the content of Figure 3.1.2 has been saved in the build directory LIME_DEBUG_BUILD under the name "lime_cmake_configure" with executable permissions, the following steps first configure and then build LIME and all needed Trilinos libraries:

```
./lime_cmake_configure
make
```

Note that the invocation of make could include the -j options for parallel compilation.

3.1.3 Run LIME Examples

Once the build completes, it is now possible to run the example problems. This can be done conveniently using ctest as follows:

```
cd LIMEExt/LIME
ctest
```

Individual examples can be run via options to ctest or manually by going into the appropriate subdirectory and using the command line options (if any) contained in the corresponding "CTestTestfile.cmake" file, eg from the top-level build directory,

```
cd LIMEExt/LIME/examples/supersimple
./LIME_lime_mpd0.exe
```

If you encounter difficulties with any of the steps, please contact

limeext-developers@software.sandia.gov with your questions or issues.

Chapter 4

Using LIME to create Multiphysics Applications

Perhaps the best way to describe and learn how to use LIME is to actually go through the steps of creating a multiphysics application. The objective of this chapter is to provide a step-by-step description of this process using several illustrative examples.

We begin with three very simple physics problems that form the basis for three "Super-Simple" computer codes. Although each problem can be viewed as a stand-alone physics problem by itself, each can also be viewed as a separate component of more complex coupled multiphysics problems. Figure 4.1 illustrates the following three SuperSimple physics problems.

1. 1-D thermal conduction through a slab with a convective heat transfer boundary condition on one side and a radiation heat transfer boundary condition on the other.
2. Attenuation of a 1-D mono-energetic flux of neutrons through a slab where the surface intensity is specified on one side and the absorption cross-section is a strong function of temperature.
3. Heat transfer from a thin highly conductive wall where the lumped mass approximation is valid.

The chapter is organized into sections according to the major steps that we typically follow when creating a new multiphysics application with LIME.

Section 4.1 describes the physics equations and theory associated with each problem, and then reviews three small computer codes, written in Fortran 90, that model each of the SuperSimple physics problems. This typifies the beginning point that a potential LIME user will usually start with.

Section 4.2 explains two things;

1. How each code is refactored so that it can be run in stand-alone mode under LIME, and
2. How to write the customized software (Multiphysics Driver and Model Evaluator) needed to wrap each code as a stand-alone application running under LIME.

Wrapping each code as a stand-alone application under LIME does not provide any additional capability, but is a useful step to insure that no errors are introduced while preparing to create a multiphysics application. Results produced by a refactored code running as a stand-alone application under LIME should identically match the results produced by the original code described in Section 4.1.

Section 4.3 explains and shows how several different multiphysics applications are created using these three codes. These include several different combinations and problem configurations for two-code applications, and also a three-code application which couples all three codes.

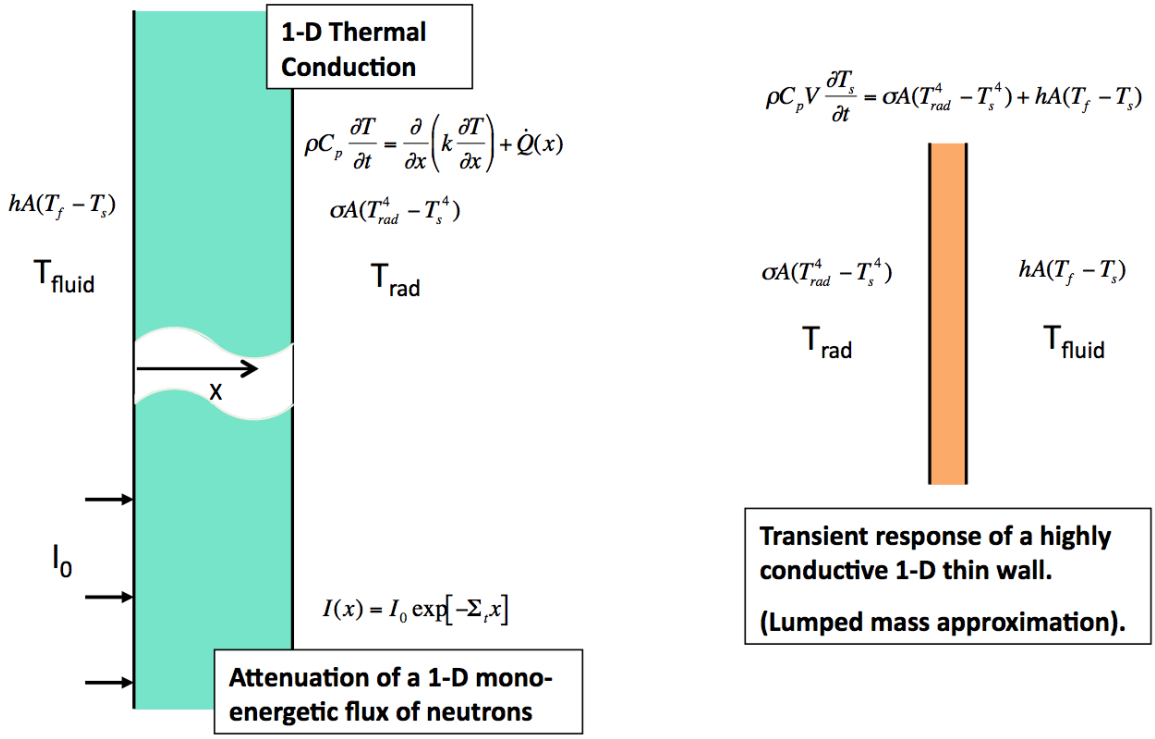


Figure 4.1. Illustration of three "super simple" physics problems.

4.1 The SuperSimple Code Suite: Theory and F90 Code

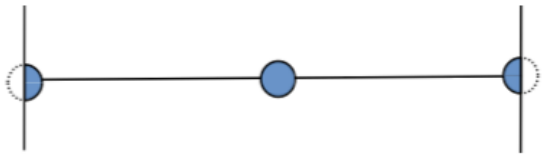
4.1.1 SuperSimple 1D Conduction (ss_con1d code)

The physics equation being approximated in ss_con1d is a transient one-dimensional conservation of energy equation that can be expressed as follows.

$$\rho C_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(k \frac{\partial T}{\partial x}\right) + \dot{Q} \tag{4.1}$$

where T is the temperature, C_p is the specific heat, ρ is the density, k is the thermal conductivity and $\dot{Q}$ is an energy source term.

In ss_con1d we assume constant material properties and solve a simple fully implicit finite-difference numerical approximation of equation 4.1 on a 1-D discretized mesh. The meshing option chosen locates boundary nodes on the surface. For a 3 node case the 1-D mesh can be illustrated like this:



In this formulation the discrete equation for an inner node can be expressed as

$$\frac{T_i - T_i^{n-1}}{dt} = \frac{k}{\rho C_p (dx)^2}\left(T_{i+1} + T_{i-1} - 2T_i\right) + \frac{1}{\rho C_p}\dot{Q}_i \tag{4.2}$$

where dt is the time step size, dx is the node spacing, and the n-1 superscript denotes temperature at the previous time step.

At nodes located on wall boundaries we assume that the boundary heat fluxes (q_{rt} and q_{lft}) are either specified, or can be represented by a simple relationship. Then a discrete equation for each wall boundary node can be written as follows.

Right side:

$$\frac{T_i - T_i^{n-1}}{dt} = \frac{k}{0.5\rho C_p (dx)^2}\left(T_{i-1} - T_i\right) - \frac{k}{0.5\rho C_p dx} q_{rt} + \frac{1}{\rho C_p}\dot{Q}_i \tag{4.3}$$

Left side:

$$\frac{T_i - T_i^{n-1}}{dt} = \frac{k}{0.5\rho C_p dx} q_{lft} - \frac{k}{0.5\rho C_p (dx)^2}\left(T_i - T_{i+1}\right) + \frac{1}{\rho C_p}\dot{Q}_i \tag{4.4}$$

Engineering approximations for surface boundary conditions are often represented as due to "convection" or "radiation", or some combination of these two. For these two situations the heat flux "q" at a boundary node i take the following form:

Convection: $q = h(T_{fluid} - T_i)$ where the convective heat transfer coefficient h and the fluid temperature T_{fluid} are specified.

Radiation: $q = \epsilon\sigma(T_{rad}^4 - T_i^4)$ where ϵ is the emissivity, σ is the Stefan-Boltzmann constant, and T_{rad} is a far-field temperature seen by the surface.

In ss_con1d, the left side is modeled with a convective heat flux boundary condition and the right side with a radiation heat transfer boundary condition.

Figure 4.2 illustrates the three-node discretization of the SuperSimple 1-D conduction problem solved in the ss_con1d code, and shows the 1-D energy equation being approximated.

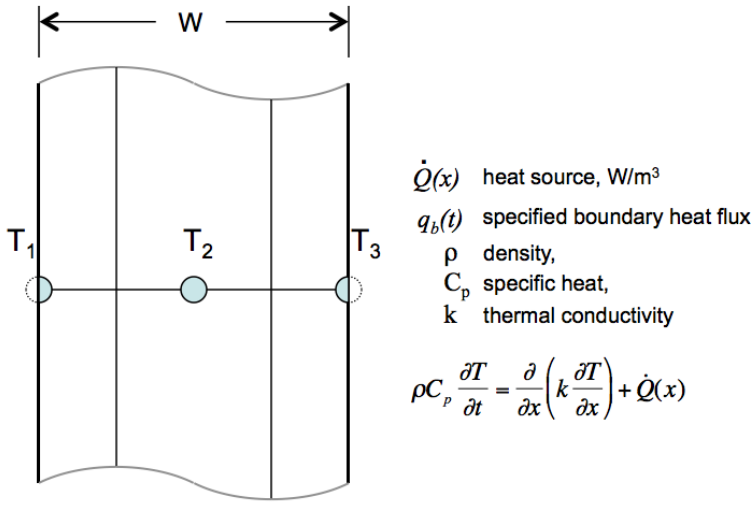


Figure 4.2. Three-node discretization of the SuperSimple 1-D conduction problem.

A complete listing of the ss_con1d Fortran 90 code written to solve this problem as described above is listed in appendix A.1. The code is approximately 110 lines long (including comments) and purposely written in a plain sequential fashion without any subroutines. In this form the code cannot be directly coupled into a LIME multiphysics application. In a subsequent section we show how the original version of ss_con1d can be easily refactored (without changing the solution) to enable it to be coupled with LIME.

4.1.2 SuperSimple 1D single-energy Neutronics (the ss_neutron code)

The idealized physics equation being approximated in ss_neutron represents the flow of mono-energetic and unidirectional neutrons through a one-dimension domain. The only mechanism

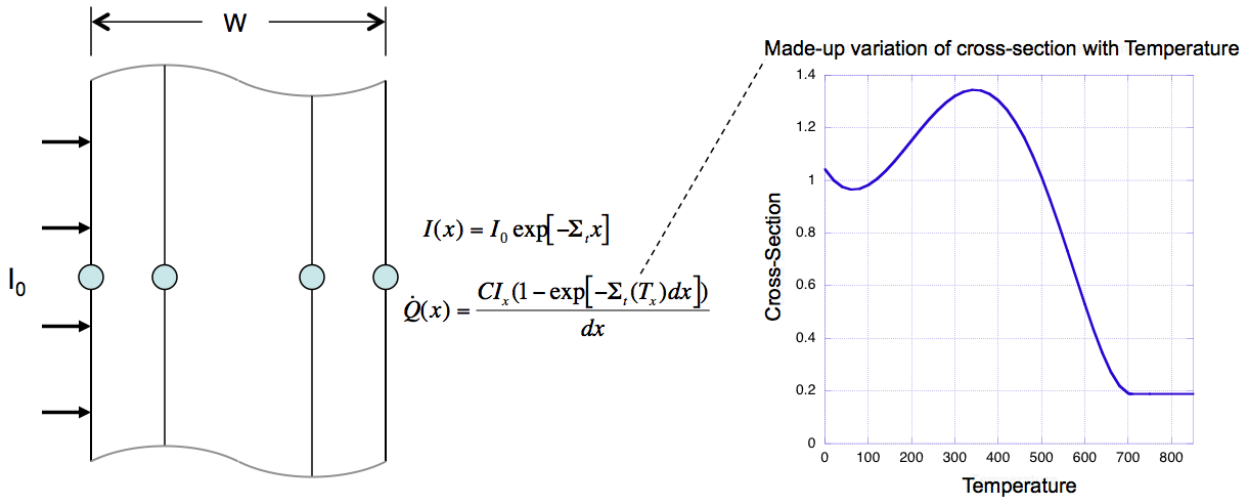


Figure 4.3. Discretization of the SuperSimple 1-D neutron-flux attenuation problem.

being modeled that affects the neutrons is neutron absorption. For this simple case we can write

$$I(x) = I_0 exp\left[-\Sigma_t x\right] \tag{4.5}$$

where

I intensity (neutrons per m^2 per sec), and

Σ_t macroscopic total cross section (1/m)

In ss_neutron we assume that the cross section Σ_t is a strong function of temperature. For illustration purposes we have made up the following functional dependence of the (normalized) cross section on temperature. It has no physical basis.

IF $T > 705 \quad \Sigma_t = 0.189, \qquad$ ELSE

$$\Sigma_t(T) = 1.042 - 2.55x10^{-3}T + 2.36x10^{-5}T^2 - 4.08x10^{-8}T^3 + 2.86x10^{-17}T^6 \tag{4.6}$$

A discrete formula that approximates equation 4.5 over the domain x to $x + dx$ can be written as follows

$$I_{x+dx} = I_x exp\left[-\Sigma_t(T_{x+dx/2})dx\right] \tag{4.7}$$

We also define an equation for relating the rate of neutron absorption to an energy release, $\dot{Q}$. This models the physical process of heat release associated with the absorption of neutrons between location x and $x+dx$ as directly proportional to the difference in neutron intensity., i.e.

$$\dot{Q}_{x+dx/2} = \frac{C\left(I_{x+dx} - I_x\right)}{dx} watts/m^3 \tag{4.8}$$

where the constant C has units of (Joules/neutron).

Figure 4.3 illustrates the simple discretization of the SuperSimple 1-D neutronics problem solved in the ss_neutron code, and shows a plot of the made-up variation of cross-section with temperature that is used in the code. Note that the discretization chosen computes neutron intensities at node locations that match the cell boundaries in the ss_con1d code previously described. This was done for convenience so that energy sources calculated here can be shared without the need for interpolation when later illustrating how to couple the codes.

A complete listing of the ss_neutron Fortran 90 code written to solve this problem as described above is listed in appendix A.2. The code is approximately 70 lines long (including comments) and purposely written in a plain sequential fashion without any subroutines. In this form the code cannot be directly coupled into a LIME multiphysics application. In a subsequent section we show how the original version of ss_neutron can be easily refactored (without changing the solution) to enable it to be coupled with LIME.

4.1.3 SuperSimple 1D thin wall (ss_thinwall Code)

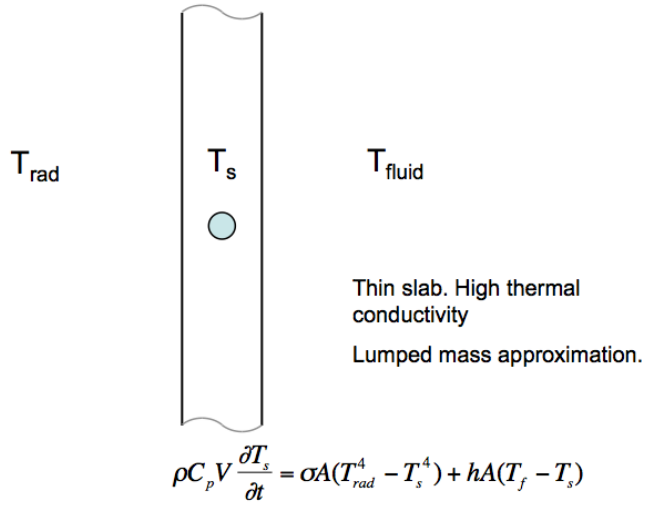


Figure 4.4. Lumped-mass (single-node) discretization of the SuperSimple thin-wall heat-transfer problem

The physics approximated in ss_thinwall is the transient thermal response of a highly conductive thin wall subject to a radiation boundary condition on the left side and a convective boundary condition on the right side. Because the thermal conductivity is high and the wall is thin, the internal thermal resistance is small relative to the convective and radiative resistance to heat flow at the boundaries. Therefore the temperature of the thin wall can be approximated as uniform in the following energy balance equation.

$$\rho C_p V \frac{\partial T}{\partial t} = \epsilon \sigma A (T_{rad}^4 - T^4) + hA(T_{fluid} - T) \tag{4.9}$$

where T is the temperature, ρ is the density, C_p is the specific heat, V is the volume, ϵ is the surface emissivity, σ is the Stefan-Boltzmann constant, A is the surface area, T_{rad} is a far-field temperature seen by the surface, h is a convective heat transfer coefficient, and T_f is a bulk fluid temperature near the surface.

Equation 4.9 is integrated in ss_thinwall using a simple first-order Euler implicit time integration scheme, which can be written as follows.

$$T = T^{n-1} + \left(\frac{\sigma A dt}{\rho C_p V}\right)(T_{rad}^4 - T^4) + \left(\frac{hAdt}{\rho C_p V}\right)(T_{fluid} - T) \tag{4.10}$$

Because the solution scheme used in ss_thinwall is based on Newtons method, we choose to rearrange the discrete equation to be in residual form as follows.

$$R(T) = T + C_3 T^4 - \frac{T^{n-1} + C_1 T_{rad}^4 + C_2 T_{fluid}}{1 + C_2} = 0 \tag{4.11}$$

where for convenience we have defined $C_1 = \frac{\sigma A dt}{\rho C_p V}$, $C_2 = \frac{hAdt}{\rho C_p V}$, and $C_3 = \frac{C_1}{1+C_2}$.

As explained earlier in section 2.2.2, Newtons method requires the derivative of the residual with respect to the state variable being solved for. The derivative of the residual with respect to T in equation 4.11 is.

$$\frac{\partial R}{\partial T} = 1 + 4C_3 T^3 \tag{4.12}$$

Therefore the Newton-based iteration scheme to be applied in the code ss_thinwall can be written as

$$T^{i+1} = T^i - \left(\frac{\partial R}{\partial T}\right)^{-1} R^i = T^i - \left(\frac{1}{1 + 4C_3 T^3}\right) R^i \tag{4.13}$$

A complete listing of the ss_thinwall Fortran 90 code written to solve this problem as described above is listed in appendix A.3. The code is approximately 90 lines long (including comments) and purposely written in a plain sequential fashion without any subroutines. In this form the code cannot be directly coupled into a LIME multiphysics application. In a subsequent section we show how the original version of ss_thinwall can be easily refactored (without changing the solution) to enable it to be coupled with LIME.

4.2 Refactoring the SuperSimple Code Suite for Stand-alone application within LIME

4.2.1 Code refactoring

Section 2.3.4 above describes several types of modifications that may be required in order for a physics code to interface with LIME. This section illustrates these changes by showing how each SuperSimple example code can be refactored to run in stand-alone mode or in a coupled mode under LIME.

4.2.1.1 con1d.f90

A complete listing of a refactored version of ss_con1d is listed in Appendix B.1. The refactored code consists of one F90 module, one F90 program unit, and eight F90 subroutines.

Figure 4.5 shows a partial listing of the code in order to highlight several points. Note first that for the very simplest stand-alone wrap with LIME only the first, third, forth, and fifth program units are needed. However, the code has also been separated into the additional routines indicated in order to provide the interfaces that will be needed for coupling to other codes. The program module is highlighted to show that the main driver logic has been separated into three basic parts; the initial set-up, the solve, and the final clean-up or finishing activities. As indicated, this routine must be commented out when compiling ss_con1d as a library.

Figure 4.6 highlights how the time step loop in the solve_con1d subroutine is consolidated into two parts. The first part is a subroutine that only executes the time-integration solve needed to advance the solution to the next time step. The second updates program and state variables after the successful completion of the time step. Each of these subroutines correspond to interfaces that are supported by the LIME Problem Manager when doing a time-dependent multiphysics problem. Also highlighted are the lines in the subroutine "residual_con1d" that compute the three components of the residual vector in this problem. The exact solution is found when each component of the residual vector is zero.

Figure 4.7 highlights some aspects of the "successive substitution" solution method used in ss_con1d. This method is essentially the "Seidel" version of the fixed point iteration scheme described in Section 2.2.2. These methods are very simple to implement but often require tuned relaxation factors to converge. Also noted is the use of the residual routine in the solve to determine whether the method has converged to the correct solution.

Another important point to be made when comparing the original code with the refactored code is that all output has been redirected to a file called "con1d_out". This was done by changing the integer variable "uout" to a value of 8, and revising all write statements to output to this output unit.

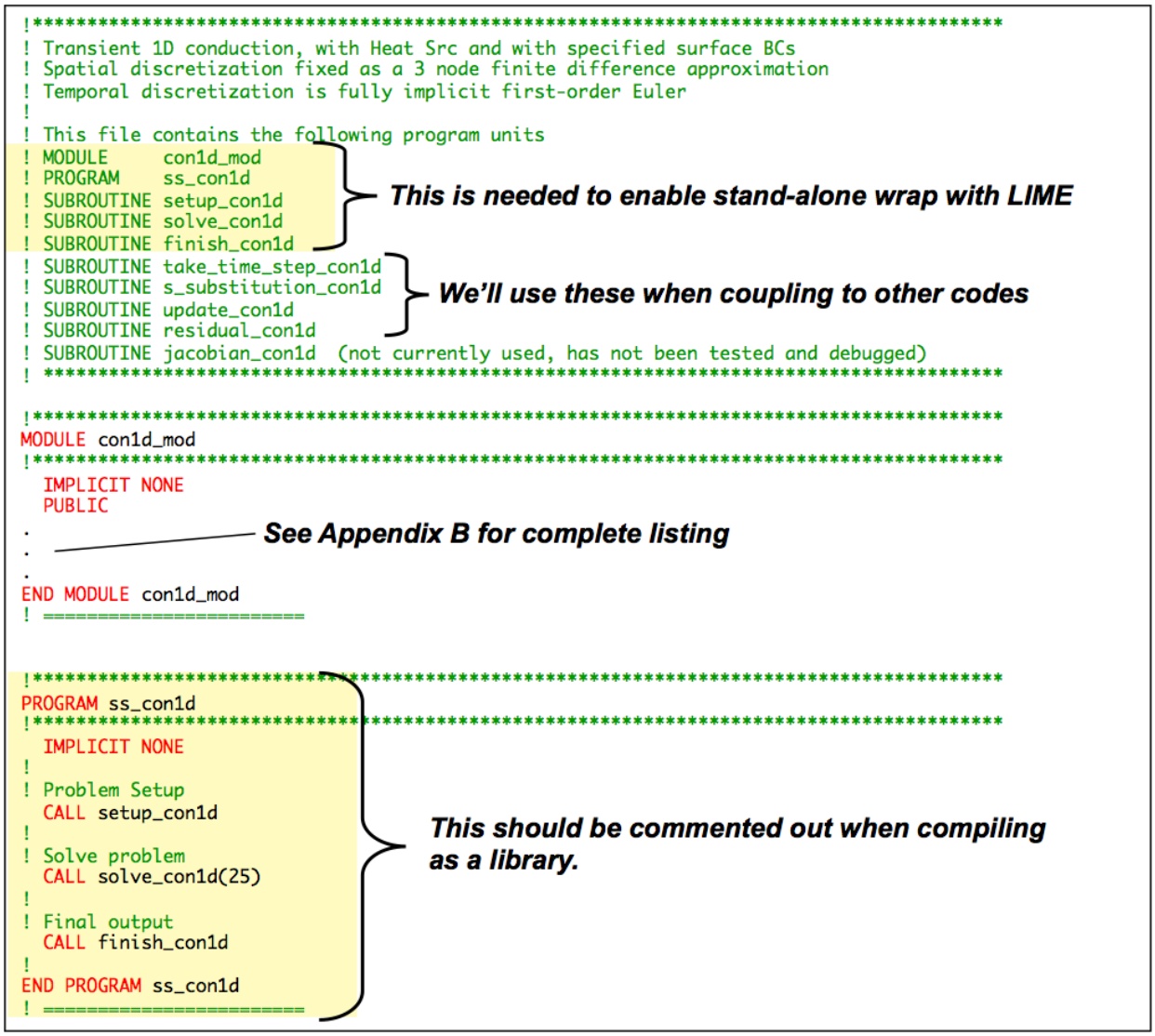


Figure 4.5. Partial code listing of refactored ss_con1d code with notations (notes on new routines).

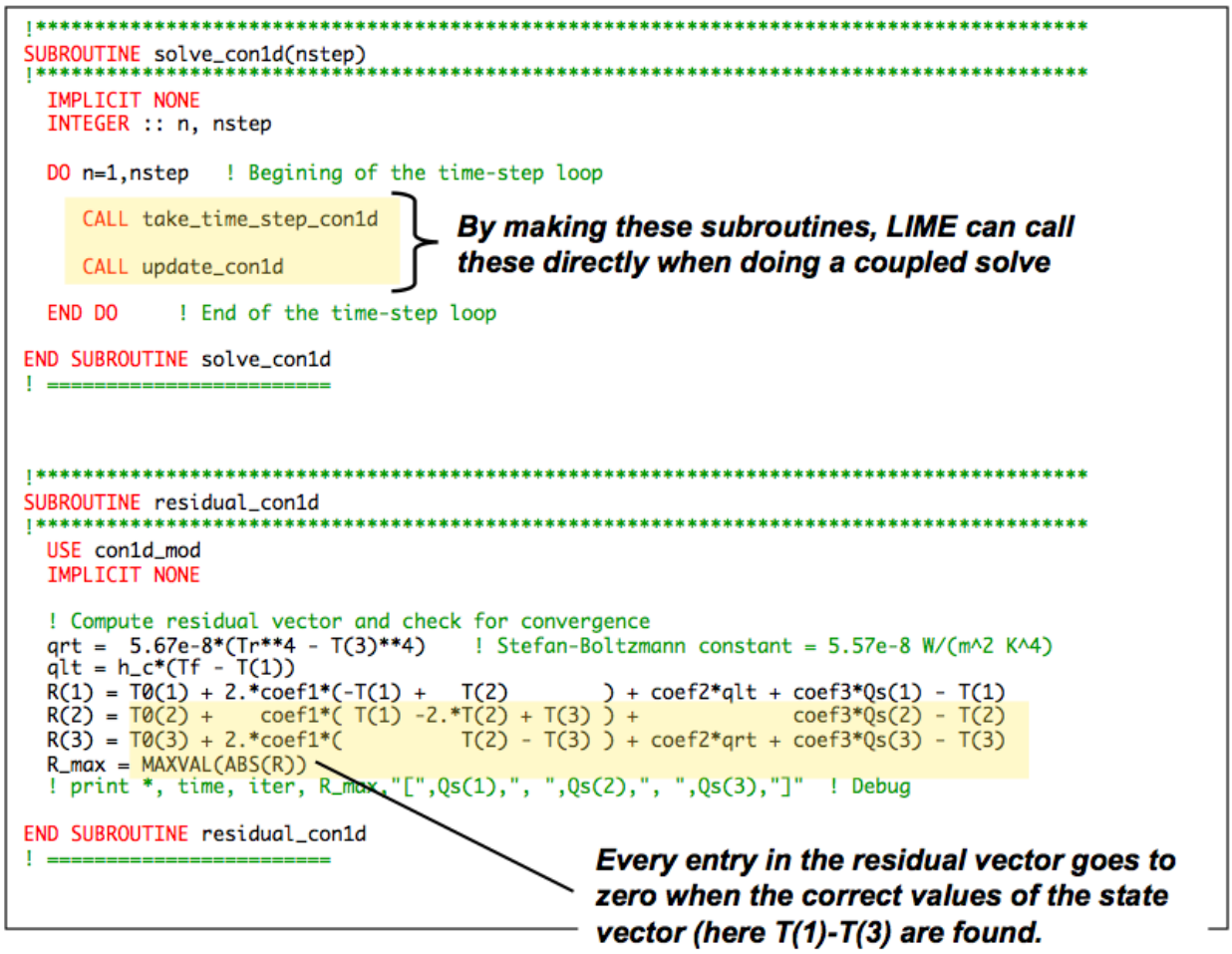
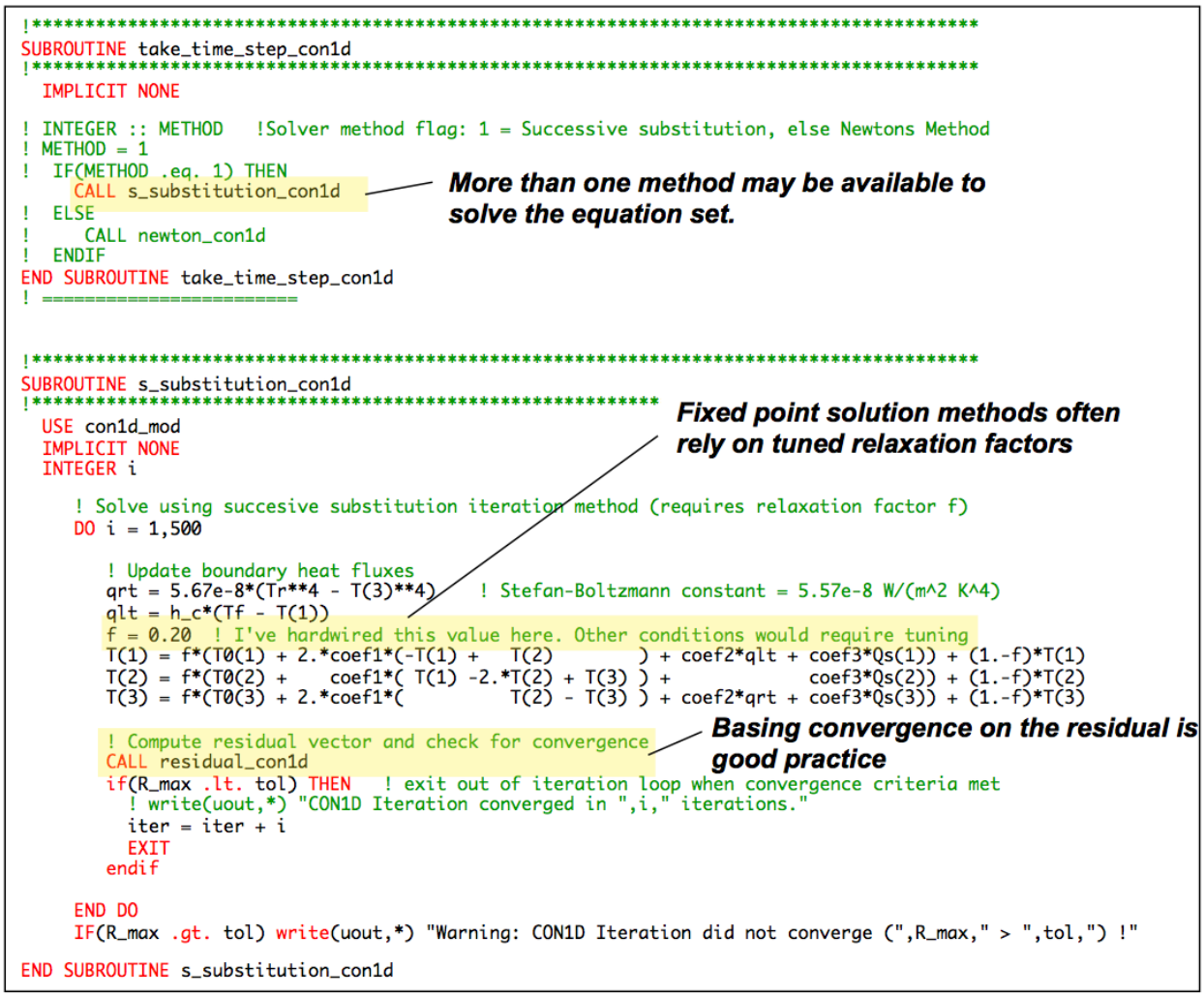


Figure 4.6. Partial code listing of refactored ss_con1d code with notations (notes on solve and residual routines).




```
!*******************************************************************************************
SUBROUTINE take_time_step_con1d
!*******************************************************************************************
  IMPLICIT NONE

! INTEGER :: METHOD   !Solver method flag: 1 = Successive substitution, else Newtons Method
! METHOD = 1
!  IF(METHOD .eq. 1) THEN
     CALL s_substitution_con1d
!  ELSE
!     CALL newton_con1d
!  ENDIF
END SUBROUTINE take_time_step_con1d
! ========================


!*******************************************************************************************
SUBROUTINE s_substitution_con1d
!*******************************************************************************************
  USE con1d_mod
  IMPLICIT NONE
  INTEGER i

     ! Solve using succesive substitution iteration method (requires relaxation factor f)
     DO i = 1,500

        ! Update boundary heat fluxes
        qrt = 5.67e-8*(Tr**4 - T(3)**4)     ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
        qlt = h_c*(Tf - T(1))
        f = 0.20  ! I've hardwired this value here. Other conditions would require tuning
        T(1) = f*(T0(1) + 2.*coef1*(-T(1) +   T(2)        ) + coef2*qlt + coef3*Qs(1)) + (1.-f)*T(1)
        T(2) = f*(T0(2) +    coef1*( T(1) -2.*T(2) + T(3) ) +             coef3*Qs(2)) + (1.-f)*T(2)
        T(3) = f*(T0(3) + 2.*coef1*(          T(2) - T(3) ) + coef2*qrt + coef3*Qs(3)) + (1.-f)*T(3)

        ! Compute residual vector and check for convergence
        CALL residual_con1d
        if(R_max .lt. tol) THEN   ! exit out of iteration loop when convergence criteria met
          ! write(uout,*) "CON1D Iteration converged in ",i," iterations."
          iter = iter + i
          EXIT
        endif

     END DO
     IF(R_max .gt. tol) write(uout,*) "Warning: CON1D Iteration did not converge (",R_max," > ",tol,") !"

END SUBROUTINE s_substitution_con1d
```

Figure 4.7. Partial code listing of refactored ss_con1d code with notations (notes on solution method used).

4.2.1.2 neutron.f90

A complete listing of a refactored version of ss_neutron is listed in Appendix B.2. The refactored code consists of one F90 module, one F90 program unit, and three F90 subroutines. When compiling as a library for use with LIME, the program module is commented out.

The ss_neutron code is a good example of a "response only" model. By this we mean that the code can be viewed as taking certain values (in this case a set of temperatures) as input, and then responding with output (in this case a set of energy sources) by directly solving its equations. In LIME, codes that represent "response only" models are often coupled using the non-linear elimination approach described in Chapter 2. For such models the main driver logic need only be separated into three basic parts; the initial set-up, the solve, and the final clean-up or finishing activities.

Note that in contrast to the original code, all output has been redirected to a file called "neutron_out". This was done by changing the integer variable "uout" to a value of 9, and revising all write statements to output to this output unit.

4.2.1.3 thinwall.f90

A complete listing of a refactored version of ss_thinwall is listed in Appendix B.3. The refactored code consists of one F90 module, one F90 program unit, and six F90 subroutines. When compiling as a library for use with LIME, the program module is commented out.

As with the other refactored SuperSimple codes, the main program unit consists of only three subroutine calls that correspond to the initial set-up, the solve, and the final clean-up or finishing activities. This problem is a transient problem, so the time step loop in the solve_thinwall subroutine is consolidated into two parts. The first part is a subroutine that only executes the time-integration solve needed to advance the solution to the next time step. The second updates program and state variables after the successful completion of the time step. Each of these subroutines correspond to interfaces that are supported by the LIME Problem Manager when doing a time-dependent multiphysics problem.

Because the original thinwall.f90 problem only has one unknown (the temperature T), the scalar quantities T and R are changed to arrays T(1) and R(1)), and an integer "n_vars" is created and set equal to 1. This is needed to facilitate the array based interface to LIME when state variables and residuals are passed up to LIME.

The solution method implemented in time_step_thinwall is based on Newton's method as described in Section 2. Because there is only 1 unknown, the Jacobian matrix reduces to a single scalar value, and finding the inverse requires no additional calculations. However, the form of this solution method is the same here as it would be if there were a large number of unknowns.

Although computing the residual is also a trivial calculation, two separate routines that

both perform the same function are written to illustrate the concept of a "thread safe" subroutine. In subroutine "residual_thinwall" the F90 module "thinwall_mod" is used to access the variables needed to compute the residual. In contrast, subroutine "residual2_thinwall" passes all variables required for computing the residual through the subroutine parameter list. The issue with thread safety is that data on the stack is, by definition, thread safe whereas accessing data in, for example, an F90 module could be unsafe if two or more threads were trying to access the data at the same time, and at least one of those accesses was writing the data. Using the stack avoids data access problems because each running thread has its own stack, hence its own copy of the data. Thus, when refactoring codes to interface with LIME, one may consider writing thread safe subroutines if doing so is possible.

Note that in contrast to the original code, all output has been redirected to a file called "thinwall_out". This was done by creating an integer variable "uout," setting it to a value of 11, and revising all write statements to output to this output unit.

4.2.2 Writing a Stand-alone Driver and Model Evaluator

This section explains how to write the customized software (Multiphysics Driver and Model Evaluator) needed to wrap each SuperSimple example code as a stand-alone application running under LIME. Wrapping each code as a stand-alone application under LIME does not provide any additional capability, but is a useful step to insure that no errors are introduced while preparing to create a multiphysics application. Results produced by a refactored code running as a stand-alone application under LIME should identically match the results produced by the original code described in Section 4.1.

4.2.2.1 con1d.f90

A complete listing for the stand-alone "driver" file for ss_con1d is given in Appendix C.1.

A complete listing of the stand-alone Model Evaluator for ss_con1d is given in Appendix C.2, and consists of two files, the C++ source file and the corresponding header file.

Figure 4.8 is an annotated listing of the first half of the stand-alone driver for ss_con1d. This figure highlights that the driver file must include the header file for the corresponding Model Evaluator. It also points out that we have written this driver using the "try" construct in C++ so that any unexpected exceptions that might occur can be handled cleanly.

Figure 4.9 is an annotated listing of the second half of the stand-alone driver for ss_con1d. This figure highlights how the Model Evaluator for ss_con1d is registered with the Problem Manager and notes that currently there are some Trilinos-specific implementation details that are used.

Figure 4.10 lists the C++ source code that a user would write for a simple stand-alone Model Evaluator for ss_con1d. This file shows how the Fortran routines contained in the

ss_con1d physics codes are referenced and used. Note the content of the "extern C" block and comments immediately above. When coupling codes written in Fortran, a problem with what is called "name mangling" must be addressed to deal with the different ways that different C++ compilers interface with Fortran subroutines and data modules. LIME has adopted a fairly general solution to this problem so that the user/developer does not need to worry about customizing the Model Evaluators for a specific compiler.

Figure 4.11 lists the associated header file that a user would write for a simple stand-alone Model Evaluator for ss_con1d. Currently, this is where a user specifies that the physics code can support a stand-alone solve. So how this is done is highlighted in this figure.

```
// ************************************************************************************************
//
// con1d_sad.cpp
// This is the LIME stand-alone driver for the super-simple con1d code
//
// ************************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d physics
#include "CON1D_ModelEval.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ------------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int rc = 0;  // return code will be set to a non zero value on errors

  try {     // the "try" construct is used to allow for catching "exceptions" that might
            // unexpectedly occur.  They can occur at any point in the program's
            // call stack so our exception handling policy is to catch all thrown excptions
            // here in main, output a diagnostic message, and terminate the program cleanly.
```

Must include the header file the model evaluator

Note on the "try" construct

Figure 4.8. Annotated listing of stand-alone Driver for ss_con1d. Part 1

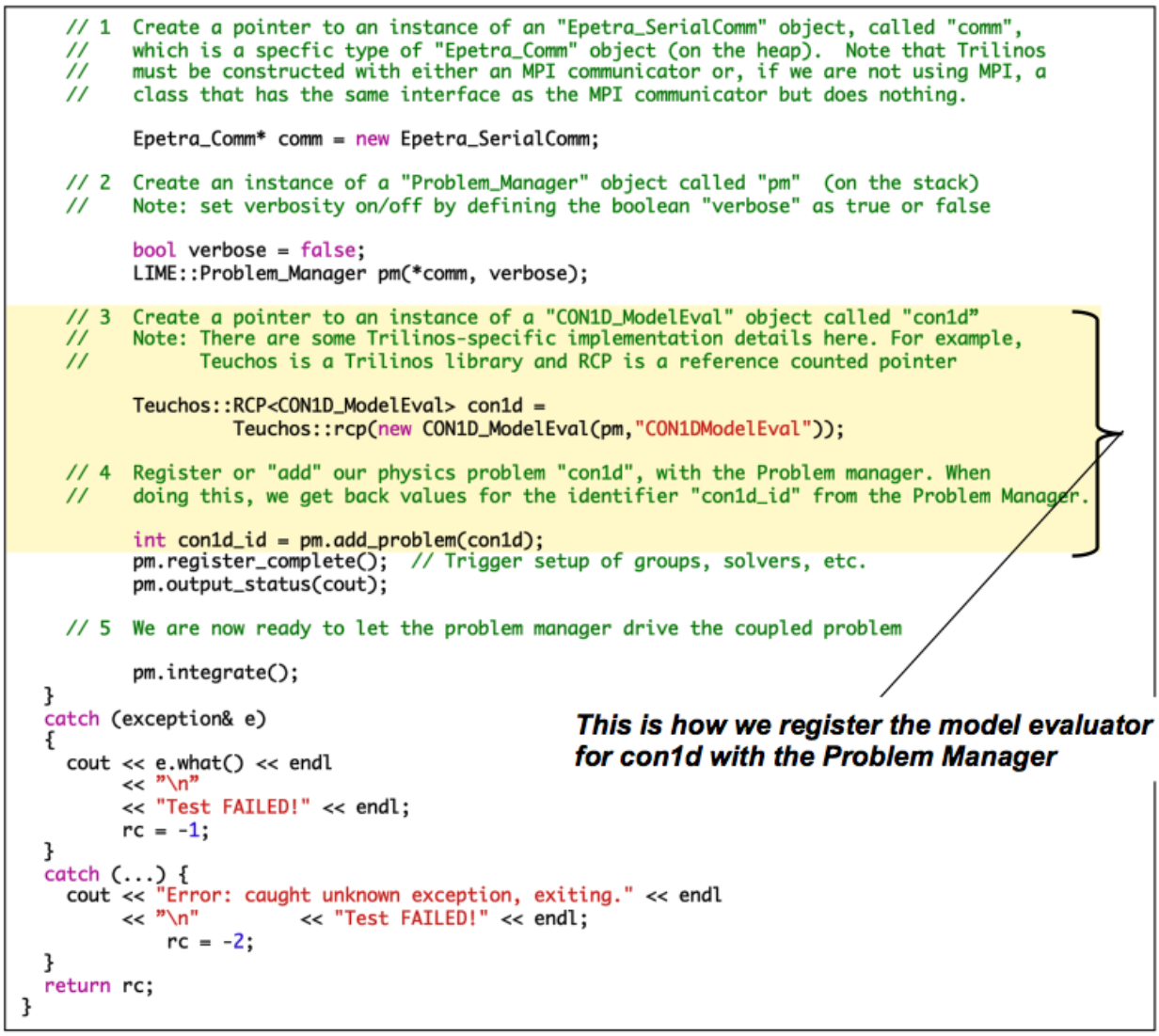


Figure 4.9. Annotated listing of stand-alone Driver for ss_con1d. Part 2

```
// ********************************************************************************************
// CON1D_ModelEval.cpp
// This is a LIME model evaluator for a standalone wrap of the ss_con1d.f90 code
// ********************************************************************************************
//
#include "CON1D_ModelEval.hpp"
#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using ModelLIME::Problem_Manager;

// Data from fortran routines are referenced using an extern C block as
// shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

 extern "C" {
   #define setup_con1d  LIME_MANGLING_GLOBAL(setup_con1d, SETUP_CON1D)
   #define solve_con1d  LIME_MANGLING_GLOBAL(solve_con1d, SOLVE_CON1D)
   #define finish_con1d LIME_MANGLING_GLOBAL(finish_con1d, FINISH_CON1D)

    void setup_con1d();
    void solve_con1d(int*);
    void finish_con1d();
 }

//------------CON1D_ModelEval C++ constructor ----------------------------------------

CON1D_ModelEval::CON1D_ModelEval(LIME::Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name)
{
  setup_con1d();           // This calls the con1d code's "setup_con1d" routine
}

//------------CON1D_ModelEval solve_standalone ---------------------------------------

void CON1D_ModelEval::solve_standalone()
{
int nstep=25;
    solve_con1d(&nstep);   // This calls the con1d code's "setup_con1d" routine
}

//------------CON1D_ModelEval C++ destructor -----------------------------------------

CON1D_ModelEval::~CON1D_ModelEval()
{
   finish_con1d();         // This calls the con1d code's "setup_con1d" routine
}
```

Figure 4.10. Annotated listing of the C++ source code for a simple stand-alone Model Evaluator for ss_con1d

```
// ********************************************************************************************
// CON1D_ModelEval.hpp
// This is a LIME model evaluator header file for stand-alone wrap of super-simple con1d code
// ********************************************************************************************
//
#ifndef LIME_EXAMPLE_CON1D_MODELEVAL_HPP
#define LIME_EXAMPLE_CON1D_MODELEVAL_HPP

#include "LIME_Model_Evaluator_API.hpp"

class CON1D_ModelEval : public LIME::Model_Evaluator
{
  public:

    CON1D_ModelEval(LIME::Problem_Manager & pm, const string & name);   // C++ constructor

    virtual ~CON1D_ModelEval();                                         // C++ destructor

    // Tells the LIME problem manger that this code supports a stand-alone solve
    virtual bool supports_standalone_solve() const { return true; }

    // The actual implimentation of "solve_standalone" will be defined in the c++ file
    virtual void solve_standalone();

};
#endif
```

Figure 4.11. Annotated listing of the C++ header file for a simple stand-alone Model Evaluator for ss_con1d

4.2.2.2 neutron.f90

A complete listing for the stand-alone "driver" file for ss_neutron is given in Appendix C.3.

A complete listing of the stand-alone Model Evaluator for ss_neutron is given in Appendix C.4, and consists of two files, the C++ source file and the corresponding header file .

4.2.2.3 thinwall.f90

A complete listing for the stand-alone "driver" file for ss_thinwall is given in Appendix C.5.

A complete listing of the stand-alone Model Evaluator for ss_thinwall is given in Appendix C.6, and consists of two files, the C++ source file and the corresponding header file.

4.3 Creating Multiphysics applications from the SuperSimple Code Suite

When creating multiphysics applications using LIME the developer/user can choose different coupling strategies. However, the choices that are viable depend on what capabilities are supported in the physics codes being coupled. This is similar to a restaurant customer who makes choices from a menu of different items that have different costs and value, but who might have medical or dietary restrictions that preclude certain foods. In like manner the developer/user of LIME must begin by choosing either a fixed point or JFNK based solver strategy, each of which has certain requirements and options. If fixed point is to be used, either a Jacobi or Seidel version can be applied, and convergence can be based on a LIME evaluation of a global residual vector or by a code by code evaluation. If JFNK is used, residuals are required from at least one of the physics codes, and either residual scaling and/or preconditioning may be needed.

The purpose of this section is to show how several different multiphysics applications can be created using the three SuperSimple physics codes, and also to illustrate some of the different ways that they can be coupled. In the subsections that follow six different examples are reviewed, each intended to illustrate a different situation or feature. The first four of these use a fixed point based solver strategy when coupling the codes and the last two use a JFNK-based solver strategy. Although many other examples could be generated, particularly for the JFNK option, these are hoped to be sufficient to get a potential user started.

1. Fixed point coupling of ss_con1d and ss_neutron where ss_neutron is treated as an elimination module (see Section 2.2.5) and where convergence is determined by querying physics code ss_con1d.

2. Fixed point coupling of ss_con1d and ss_thinwall where both physics codes are queried to determine convergence.

3. Fixed point coupling (Jacobi) of ss_con1d, ss_thinwall and ss_neutron where ss_neutron is treated as an elimination module (see Section 2.2.5) and where the physics codes determine convergence.

4. Fixed point coupling of ss_con1d and ss_thinwall where the Problem Manager determines convergence by evaluating a global residual (i.e. same physics and coupling as Example 2, but different convergence metric).

5. JFNK-based coupling of ss_con1d and ss_thinwall (i.e. same physics as examples 2 and 4, but different nonlinear solution method).

6. JFNK-based coupling of of ss_con1d, ss_thinwall and ss_neutron where ss_neutron is treated as an elimination module and where the Problem Manager determines convergence (i.e. same physics as Example 3, but different nonlinear solution method).

Examples 1-4 (our fixed-point based examples) use the Problem_Manager_setup.xml file shown below in Fig. 4.12. Note that this file is actually not required for these example problems because the "Solver Strategy" and the "Use Predictor" parameters chosen are the default values and because the maximum number of times steps is always set in the Model Evaluator (and is not restricted further here). Examples 5 and 6, which are coupled using the JFNK strategy use the Problem_Manager_setup.xml file shown in Fig. 4.13.

The parameter settings for the fixed point solver used for problems 1-4 can be seen in the Problem_Manager_setup_fp.xml file shown in Fig. 4.14. Those for the JFNK solver used for problems 5 and 6 are shown in Fig. 4.15.

```
<ParameterList>
  <Parameter name="Solver Strategy" type="string" value="FIXED_POINT"/>
  <Parameter name="Use Predictor" type="bool" value="false"/>
  <Parameter name="Max Time Steps" type="int" value="1000"/>
</ParameterList>
```

Figure 4.12. Listing of the "Problem_Manager_setup.xml" for example problems 1-4.

```
<ParameterList>
  <Parameter name="Solver Strategy" type="string" value="JFNK"/>
  <Parameter name="Use Predictor" type="bool" value="false"/>
  <Parameter name="Max Time Steps" type="int" value="1000"/>
</ParameterList>
```

Figure 4.13. Listing of the "Problem_Manager_setup.xml" for example problems 5 and 6.

```
<ParameterList>
  <Parameter name="Solve Mode" type="string" value="Jacobi"/>
  <Parameter name="Maximum Iterations" type="int" value="20"/>
  <Parameter name="Absolute Tolerance" type="double" value="2e-03"/>
</ParameterList>
```

Figure 4.14. Listing of the "Problem_Manager_setup_fp.xml" for example problems 1-4.

```
<ParameterList>
  <Parameter name="Preconditioner Operator" type="string" value="None"/>
  <Parameter name="Maximum Iterations" type="int" value="3"/>
  <Parameter name="Absolute Tolerance" type="double" value="2e-03"/>
</ParameterList>
```

Figure 4.15. Listing of the "Problem_Manager_setup_jfnk.xml" for example problems 5 and 6.

4.3.1 Example 1: Fixed point coupling of ss_con1d and ss_neutron with elimination

In this example we couple ss_con1d and ss_neutron, where ss_neutron is treated as an elimination module and where convergence at each time step is determined by querying physics code ss_con1d. Because the equations applied in ss_neutron provide a direct solution for the local power, treating ss_neutron in this way is natural because in this setting ss_neutron is effectively a response only model. We note that since only two codes are being coupled, this choice is algorithmically equivalent to a "Seidel" type fixed point solution. However, if the problem was framed in this way (i.e. without defining ss_neutron as an elimination module) an additional interface that defined convergence would need to be written and defined for ss_neutron.

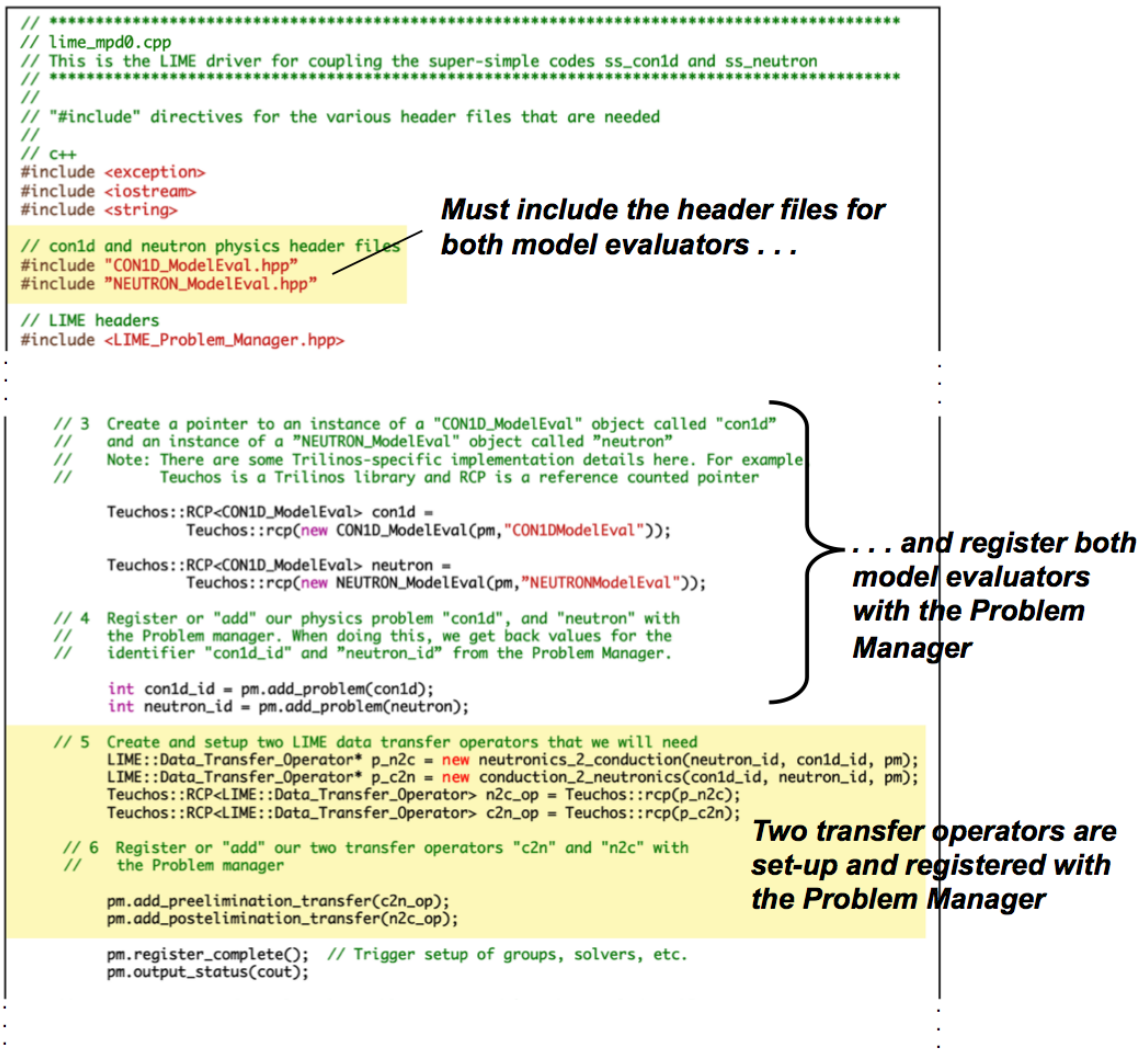


Figure 4.16. Annotated partial listing of the Multiphysics Driver for Example 1.

A complete listing of the Multiphysics Driver file for the Example 1 application is given in Appendix D.1.1. Figure 4.16 provides a partial listing that highlights several important points. First, we note that the structure and content of the Multiphysics Driver is very similar to those presented earlier for the stand-alone applications. The first major difference is simply that we now have two physics codes, thus header files for both Model Evaluator files must be "#included" and they must both be properly registered with the Problem Manager. The second major difference is that the transfer operators that will be used must be setup and registered as shown. The actual implementation of the transfer operators will be found in the revised Model Evaluator for ss_neutron.

A complete listing of the Model Evaluator for ss_con1d used in the Example 1 application is given in Appendix D.1.2, and consists of two files, the C++ source file and the corresponding header file. Figure 4.17 gives a partial listing of the the C++ source file that highlights several important points. First, there are additional routines and data needed for the coupled problem, and these are defined in the extern C block. In addition, there are several more Problem_Manager/Model_Evaluator interfaces that must be implemented for this problem. These are also reflected in the highlighted section in Figure 4.18, which lists the associated C++ header file.

A complete listing of the Model Evaluator for ss_neutron used in the Example 1 application is given in Appendix D.1.3, and consists of two files, the C++ source file and the corresponding header file.

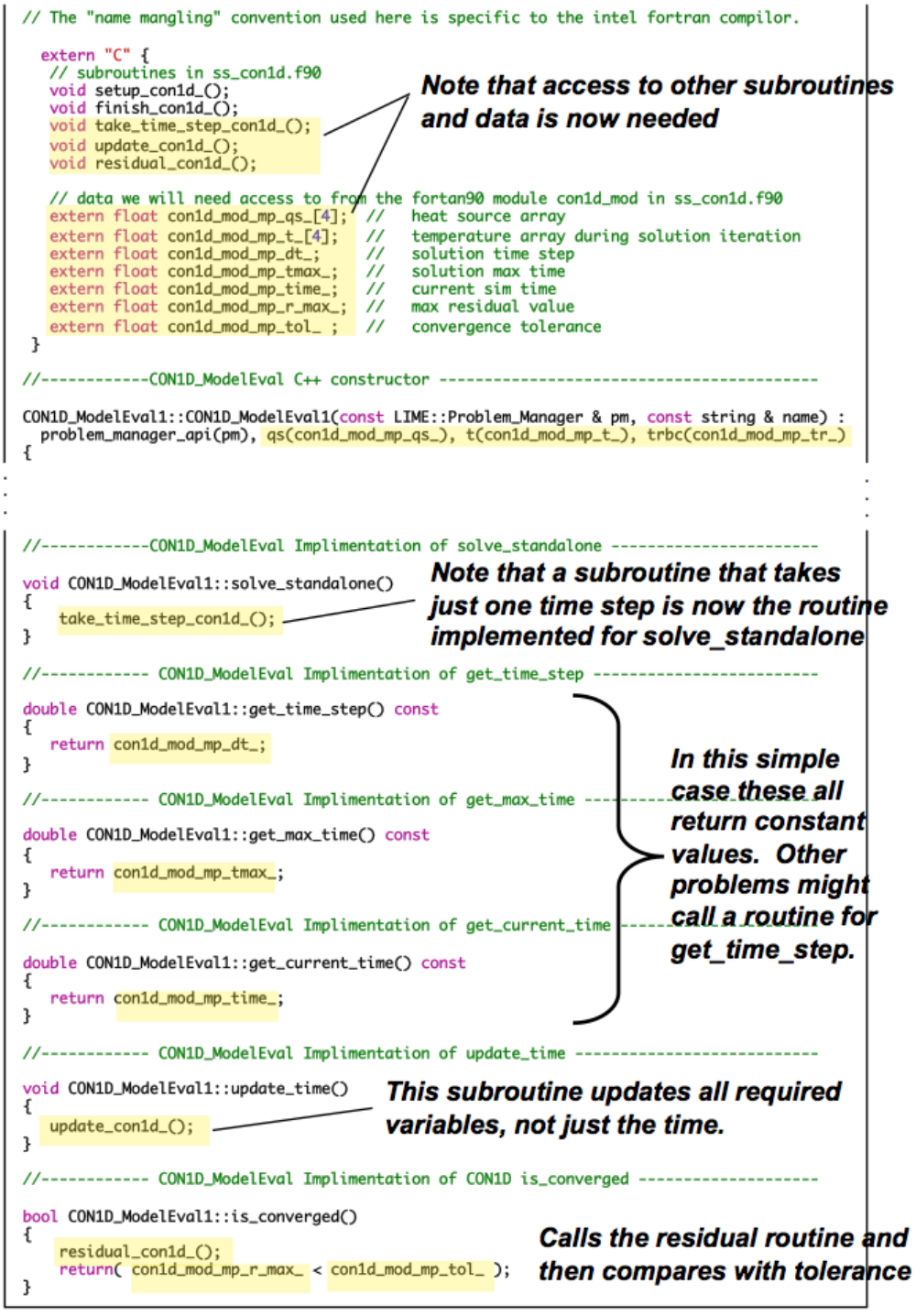


Figure 4.17. Annotated partial listing of the C++ source code for the ss_con1d Model Evaluator for Example 1.

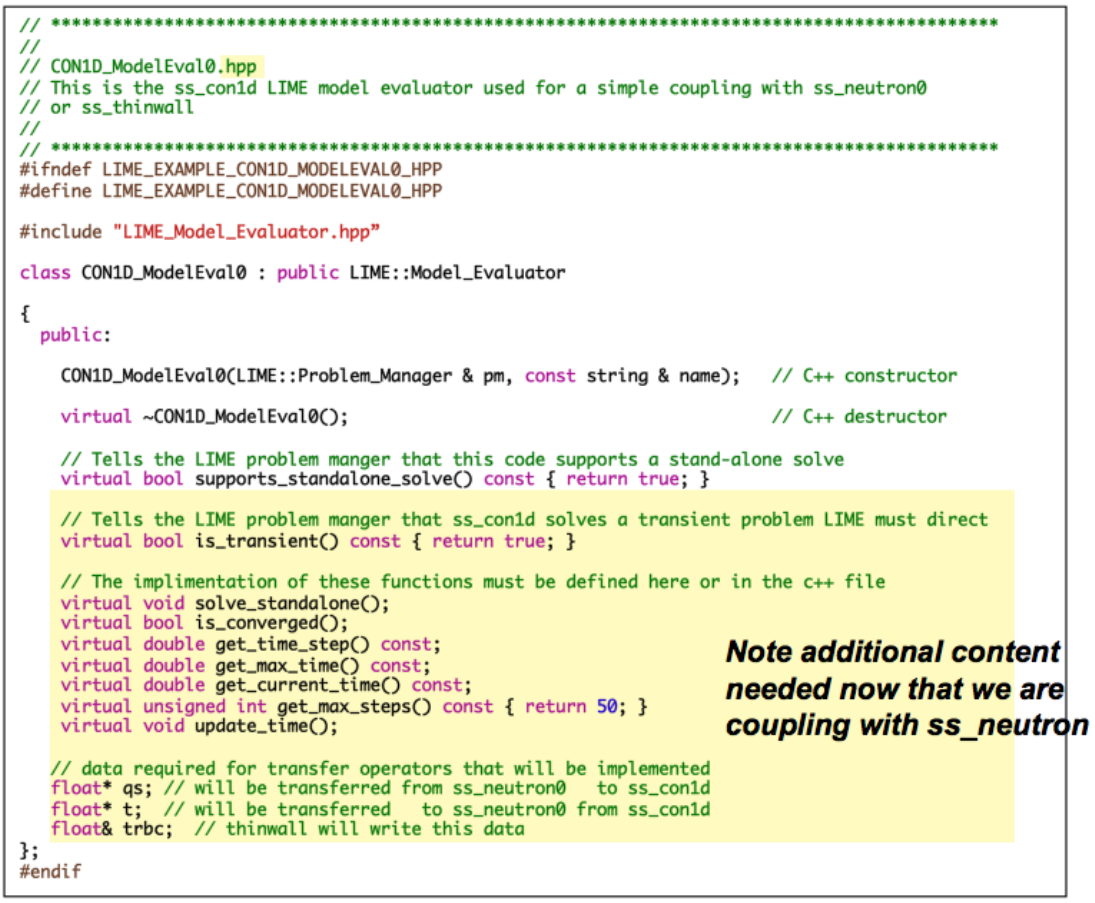


Figure 4.18. Annotated listing of the C++ header file for the ss_con1d Model Evaluator for Example 1.

Figures 4.19 and 4.20 give listings of the output files produced by each of the physics codes when running Example Problem 1.

```
time     qlt        T(1)        T(2)        T(3)         qrt       ebal      iter    R_max
   0.0   0.00E+00   3.000E+02   3.000E+02   3.000E+02    0.00E+00  0.00E+00     0    0.00E+00
   2.0  -3.11E+00   3.811E+02   3.709E+02   3.381E+02   -2.82E+02  3.04E-01    51    1.53E-04
   4.0  -8.66E+00   4.366E+02   4.184E+02   3.577E+02   -4.69E+02  5.29E-01    74    5.80E-04
   6.0  -1.22E+01   4.717E+02   4.480E+02   3.686E+02   -5.87E+02  6.95E-01    80    1.80E-03
   8.0  -1.43E+01   4.931E+02   4.659E+02   3.747E+02   -6.59E+02  8.10E-01    79    1.40E-03
  10.0  -1.56E+01   5.057E+02   4.766E+02   3.783E+02   -7.02E+02  8.85E-01    77    1.34E-03
  12.0  -1.63E+01   5.132E+02   4.828E+02   3.803E+02   -7.27E+02  9.31E-01    70    1.34E-03
  14.0  -1.67E+01   5.175E+02   4.864E+02   3.815E+02   -7.42E+02  9.60E-01    63    9.16E-04
  16.0  -1.70E+01   5.200E+02   4.885E+02   3.822E+02   -7.50E+02  9.76E-01    56    8.54E-04
  18.0  -1.71E+01   5.215E+02   4.897E+02   3.826E+02   -7.55E+02  9.86E-01    49    1.71E-03
  20.0  -1.72E+01   5.223E+02   4.904E+02   3.828E+02   -7.58E+02  9.92E-01    43    9.77E-04
  22.0  -1.73E+01   5.228E+02   4.908E+02   3.829E+02   -7.60E+02  9.95E-01    36    1.04E-03
  24.0  -1.73E+01   5.231E+02   4.910E+02   3.830E+02   -7.61E+02  9.97E-01    31    1.71E-03
  26.0  -1.73E+01   5.232E+02   4.912E+02   3.830E+02   -7.61E+02  9.98E-01    27    8.54E-04
  28.0  -1.73E+01   5.233E+02   4.912E+02   3.831E+02   -7.61E+02  9.99E-01    22    9.77E-04
  30.0  -1.73E+01   5.234E+02   4.913E+02   3.831E+02   -7.62E+02  9.99E-01    17    1.71E-03
  32.0  -1.73E+01   5.234E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00    14    6.71E-04
  34.0  -1.73E+01   5.234E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00    12    2.44E-04
  36.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     9    1.16E-03
  38.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     5    5.80E-04
  40.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     4    3.05E-04
  42.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     3    6.10E-04
  44.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     1    1.59E-03
  46.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     1    1.40E-03
  48.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     0    1.83E-03
  50.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     0    1.83E-03
  52.0  -1.73E+01   5.235E+02   4.913E+02   3.831E+02   -7.62E+02  1.00E+00     0    1.83E-03
Finished running program ss_con1d
```

Figure 4.19. Listing of output file "con1d_out" produced by the ss_con1d code for Example Problem 1.

```
 X            I            Q            T            TCS
0.000E+00    1.000E+00     -           5.235E+02
2.500E-02     -           3.559E+03     -           9.101E-01
5.000E-02    9.555E-01     -            -
1.000E-01     -           3.807E+03    4.913E+02    1.049E+00
1.500E-01    8.603E-01     -            -
1.750E-01     -           4.413E+03     -           1.325E+00
2.000E-01    8.052E-01     -           3.831E+02
Finished running program ss_neutron
```

Figure 4.20. Listing of output file "neutron0_out" produced by the ss_neutron0 code for Example Problem 1.

4.3.2 Example 2: Fixed point coupling of ss_con1d and ss_thinwall using local convergence checks

In this example we couple ss_con1d and ss_thinwall using a simple fixed point approach where both physics codes are queried to determine convergence. The Model Evaluator used for ss_con1d here is the same as is used in Example 1 (see Appendix D.1.2).

The Multiphysics Driver for Example 2 is listed in Appendix D.2.1. This file is identical in form to the Multiphysics Driver used for Example 1. The only differences correspond to replacing the various names associated with the ss_neutron code with those of the ss_thinwall code in the text.

The Model Evaluator for the ss_thinwall code is listed in Appendix D.2.2. Because ss_thinwall is not an elimination module (in contrast to ss_neutron in Example 1), the Model Evaluator must be written to support all of the required Problem_Manager/Model_Evaluator interfaces that will be needed. These interfaces are exactly the same ones that were needed for the ss_con1d code, but are here implemented for the ss_thinwall code. In addition, the transfer operators must be defined that enable the appropriate transfer of data from ss_con1d to ss_thinwall and also from ss_thinwall to ss_con1d.

Figures 4.21 and 4.22 give listings of the output files produced by each of the physics codes when running Example Problem 2.

```
time     qlt        T(1)        T(2)        T(3)         qrt        ebal      iter    R_max
   0.0   0.00E+00   3.000E+02   3.000E+02   3.000E+02    0.00E+00   0.00E+00     0    0.00E+00
   2.0  -1.60E-02   3.502E+02   3.452E+02   3.256E+02   -1.84E+02   3.07E-01    83   6.71E-04
   4.0  -3.81E+00   3.881E+02   3.779E+02   3.403E+02   -3.01E+02   5.08E-01    70   1.37E-03
   6.0  -6.61E+00   4.161E+02   4.014E+02   3.503E+02   -3.81E+02   6.46E-01    68   1.50E-03
   8.0  -8.65E+00   4.365E+02   4.185E+02   3.575E+02   -4.37E+02   7.42E-01    71   1.22E-03
  10.0  -1.02E+01   4.515E+02   4.310E+02   3.631E+02   -4.76E+02   8.10E-01    71   1.50E-03
  12.0  -1.13E+01   4.625E+02   4.403E+02   3.674E+02   -5.04E+02   8.59E-01    70   1.65E-03
  14.0  -1.21E+01   4.708E+02   4.473E+02   3.708E+02   -5.24E+02   8.93E-01    68   1.59E-03
  16.0  -1.27E+01   4.770E+02   4.526E+02   3.736E+02   -5.38E+02   9.18E-01    65   1.95E-03
  18.0  -1.32E+01   4.817E+02   4.567E+02   3.758E+02   -5.49E+02   9.37E-01    65   1.37E-03
  20.0  -1.35E+01   4.854E+02   4.599E+02   3.777E+02   -5.56E+02   9.50E-01    62   1.43E-03
  22.0  -1.38E+01   4.883E+02   4.625E+02   3.793E+02   -5.62E+02   9.60E-01    62   1.56E-03
  24.0  -1.41E+01   4.906E+02   4.646E+02   3.806E+02   -5.67E+02   9.68E-01    61   1.77E-03
  26.0  -1.42E+01   4.925E+02   4.662E+02   3.817E+02   -5.70E+02   9.74E-01    61   1.65E-03
  28.0  -1.44E+01   4.940E+02   4.676E+02   3.827E+02   -5.73E+02   9.78E-01    58   1.59E-03
  30.0  -1.45E+01   4.952E+02   4.688E+02   3.834E+02   -5.75E+02   9.82E-01    58   1.46E-03
  32.0  -1.46E+01   4.963E+02   4.697E+02   3.841E+02   -5.76E+02   9.85E-01    57   1.56E-03
  34.0  -1.47E+01   4.971E+02   4.705E+02   3.847E+02   -5.78E+02   9.87E-01    55   1.65E-03
  36.0  -1.48E+01   4.979E+02   4.712E+02   3.852E+02   -5.79E+02   9.89E-01    53   1.19E-03
  38.0  -1.48E+01   4.985E+02   4.718E+02   3.856E+02   -5.80E+02   9.91E-01    52   1.34E-03
  40.0  -1.49E+01   4.990E+02   4.723E+02   3.860E+02   -5.80E+02   9.92E-01    52   1.43E-03
  42.0  -1.49E+01   4.995E+02   4.727E+02   3.863E+02   -5.81E+02   9.93E-01    49   1.74E-03
  44.0  -1.50E+01   4.998E+02   4.730E+02   3.865E+02   -5.82E+02   9.94E-01    43   1.28E-03
  46.0  -1.50E+01   5.001E+02   4.733E+02   3.868E+02   -5.82E+02   9.95E-01    41   1.19E-03
  48.0  -1.50E+01   5.004E+02   4.736E+02   3.869E+02   -5.83E+02   9.96E-01    39   1.98E-03
  50.0  -1.51E+01   5.006E+02   4.738E+02   3.871E+02   -5.83E+02   9.97E-01    39   1.89E-03
  52.0  -1.51E+01   5.008E+02   4.740E+02   3.873E+02   -5.83E+02   9.97E-01    37   1.95E-03
Finished running program ss_con1d
```

Figure 4.21. Listing of output file "con1d_out" produced by the ss_con1d code for Example Problem 2.

```
 time     qrad        T          qfluid   iter    Resid
   0.0   0.00E+00   3.000E+02    0.00E+00
   2.0   1.84E+02   2.989E+02   -2.39E+02   1    1.83E-04
   4.0   3.01E+02   2.999E+02   -2.49E+02   1    1.83E-04
   6.0   3.81E+02   3.021E+02   -2.71E+02   1    4.58E-04
   8.0   4.37E+02   3.049E+02   -2.99E+02   1    7.02E-04
  10.0   4.76E+02   3.078E+02   -3.28E+02   1    8.54E-04
  12.0   5.04E+02   3.108E+02   -3.58E+02   1    9.77E-04
  14.0   5.24E+02   3.135E+02   -3.85E+02   1    1.04E-03
  16.0   5.38E+02   3.161E+02   -4.11E+02   1    2.62E-03
  18.0   5.49E+02   3.184E+02   -4.34E+02   1    2.35E-03
  20.0   5.56E+02   3.204E+02   -4.54E+02   1    2.08E-03
  22.0   5.62E+02   3.222E+02   -4.72E+02   1    1.83E-03
  24.0   5.67E+02   3.238E+02   -4.88E+02   1    1.53E-03
  26.0   5.70E+02   3.252E+02   -5.02E+02   1    1.37E-03
  28.0   5.73E+02   3.263E+02   -5.13E+02   1    1.22E-03
  30.0   5.75E+02   3.274E+02   -5.24E+02   1    1.07E-03
  32.0   5.76E+02   3.282E+02   -5.32E+02   1    9.46E-04
  34.0   5.78E+02   3.290E+02   -5.40E+02   1    7.32E-04
  36.0   5.79E+02   3.296E+02   -5.46E+02   1    6.41E-04
  38.0   5.80E+02   3.302E+02   -5.52E+02   1    5.19E-04
  40.0   5.80E+02   3.307E+02   -5.57E+02   1    4.88E-04
  42.0   5.81E+02   3.311E+02   -5.61E+02   1    4.88E-04
  44.0   5.82E+02   3.314E+02   -5.64E+02   1    4.43E-03
  46.0   5.82E+02   3.317E+02   -5.67E+02   1    3.85E-03
  48.0   5.83E+02   3.320E+02   -5.70E+02   1    3.39E-03
  50.0   5.83E+02   3.322E+02   -5.72E+02   1    2.93E-03
  52.0   5.83E+02   3.324E+02   -5.74E+02   1    2.50E-03
Finished running program ss_thinwall
```

Figure 4.22. Listing of output file "thinwall_out" produced by the ss_thinwall code for Example Problem 2.

4.3.3 Example 3: Fixed point coupling of ss_con1d, ss_thinwall and ss_neutron

In this example we couple ss_con1d, ss_thinwall and ss_neutron, where ss_neutron is treated as an elimination module (see Section 2.2.5) and where the physics codes determine convergence. The three different physical processes being coupled in this problem were illustrated earlier in Fig. 4.1. The Model Evaluators used here for each of the three SuperSimple physics codes being coupled are identical to those previously used for Example Problems 1 and 2, and thus can be found in Appendices D.1.2, D.1.3 and D.2.2. Note that the transfer operators needed for this combined problem are also the same ones used previously in Examples 1 and 2.

The Multiphysics Driver for Example 3 is listed in D.2.3.

Figures 4.23, 4.24 and 4.25 give listings of the output files produced by each of the physics codes when running Example Problem 3.

To help illustrate the affect of the coupled physics, Figure 4.26 compares the slab surface temperature as a function of time for each of the first three example problems.

```
time     qlt        T(1)        T(2)        T(3)         qrt        ebal      iter    R_max
   0.0   0.00E+00   3.000E+02   3.000E+02   3.000E+02    0.00E+00   0.00E+00     0    0.00E+00
   2.0  -3.12E+00   3.812E+02   3.710E+02   3.383E+02   -2.80E+02   3.03E-01    83    1.25E-03
   4.0  -8.70E+00   4.370E+02   4.190E+02   3.593E+02   -4.61E+02   5.21E-01    94    8.85E-04
   6.0  -1.23E+01   4.729E+02   4.497E+02   3.721E+02   -5.73E+02   6.80E-01    93    1.31E-03
   8.0  -1.45E+01   4.952E+02   4.689E+02   3.805E+02   -6.39E+02   7.91E-01    90    1.86E-03
  10.0  -1.59E+01   5.089E+02   4.809E+02   3.862E+02   -6.78E+02   8.64E-01    89    1.43E-03
  12.0  -1.68E+01   5.175E+02   4.885E+02   3.903E+02   -6.99E+02   9.11E-01    91    1.40E-03
  14.0  -1.73E+01   5.230E+02   4.935E+02   3.934E+02   -7.11E+02   9.40E-01    95    1.25E-03
  16.0  -1.77E+01   5.265E+02   4.968E+02   3.958E+02   -7.18E+02   9.59E-01    97    1.98E-03
  18.0  -1.79E+01   5.290E+02   4.992E+02   3.977E+02   -7.21E+02   9.70E-01    93    1.22E-03
  20.0  -1.81E+01   5.307E+02   5.009E+02   3.992E+02   -7.23E+02   9.78E-01    94    1.22E-03
  22.0  -1.82E+01   5.320E+02   5.022E+02   4.005E+02   -7.24E+02   9.83E-01    94    1.34E-03
  24.0  -1.83E+01   5.329E+02   5.032E+02   4.015E+02   -7.24E+02   9.87E-01    93    1.10E-03
  26.0  -1.84E+01   5.337E+02   5.040E+02   4.023E+02   -7.24E+02   9.89E-01    91    1.62E-03
  28.0  -1.84E+01   5.343E+02   5.046E+02   4.030E+02   -7.24E+02   9.91E-01    91    1.19E-03
  30.0  -1.85E+01   5.347E+02   5.051E+02   4.036E+02   -7.24E+02   9.93E-01    94    1.65E-03
  32.0  -1.85E+01   5.351E+02   5.055E+02   4.041E+02   -7.24E+02   9.94E-01    87    1.68E-03
  34.0  -1.85E+01   5.354E+02   5.059E+02   4.045E+02   -7.24E+02   9.95E-01    88    1.65E-03
  36.0  -1.86E+01   5.357E+02   5.062E+02   4.049E+02   -7.24E+02   9.96E-01    88    1.89E-03
  38.0  -1.86E+01   5.359E+02   5.064E+02   4.051E+02   -7.24E+02   9.97E-01    82    1.62E-03
  40.0  -1.86E+01   5.361E+02   5.066E+02   4.054E+02   -7.24E+02   9.97E-01    80    1.53E-03
  42.0  -1.86E+01   5.363E+02   5.068E+02   4.056E+02   -7.24E+02   9.98E-01    74    1.83E-03
  44.0  -1.86E+01   5.364E+02   5.069E+02   4.057E+02   -7.23E+02   9.98E-01    74    1.68E-03
  46.0  -1.86E+01   5.365E+02   5.070E+02   4.058E+02   -7.23E+02   9.98E-01    64    1.53E-03
  48.0  -1.87E+01   5.366E+02   5.071E+02   4.060E+02   -7.23E+02   9.99E-01    67    1.71E-03
  50.0  -1.87E+01   5.366E+02   5.072E+02   4.060E+02   -7.23E+02   9.99E-01    61    1.56E-03
  52.0  -1.87E+01   5.367E+02   5.073E+02   4.061E+02   -7.23E+02   9.99E-01    56    1.40E-03
Finished running program ss_con1d
```

Figure 4.23. Listing of output file "con1d_out" produced by the ss_con1d code for Example Problem 3.

```
 time     qrad        T          qfluid   iter    Resid
   0.0   0.00E+00   3.000E+02    0.00E+00
   2.0   2.80E+02   3.005E+02   -2.55E+02   1    2.75E-04
   4.0   4.61E+02   3.039E+02   -2.89E+02   1    6.10E-05
   6.0   5.73E+02   3.087E+02   -3.37E+02   1    2.14E-04
   8.0   6.39E+02   3.137E+02   -3.87E+02   1    7.63E-04
  10.0   6.78E+02   3.185E+02   -4.35E+02   1    1.22E-03
  12.0   6.99E+02   3.229E+02   -4.79E+02   1    1.59E-03
  14.0   7.11E+02   3.268E+02   -5.18E+02   1    1.80E-03
  16.0   7.18E+02   3.301E+02   -5.51E+02   1    2.44E-04
  18.0   7.21E+02   3.330E+02   -5.80E+02   1    2.59E-03
  20.0   7.23E+02   3.354E+02   -6.04E+02   1    2.01E-03
  22.0   7.24E+02   3.374E+02   -6.24E+02   1    1.71E-03
  24.0   7.24E+02   3.390E+02   -6.40E+02   1    1.34E-03
  26.0   7.24E+02   3.404E+02   -6.54E+02   1    1.13E-03
  28.0   7.24E+02   3.416E+02   -6.66E+02   1    9.46E-04
  30.0   7.24E+02   3.426E+02   -6.76E+02   1    8.54E-04
  32.0   7.24E+02   3.434E+02   -6.84E+02   1    6.41E-04
  34.0   7.24E+02   3.440E+02   -6.90E+02   1    5.80E-04
  36.0   7.24E+02   3.446E+02   -6.96E+02   1    3.97E-04
  38.0   7.24E+02   3.451E+02   -7.01E+02   1    3.97E-04
  40.0   7.24E+02   3.454E+02   -7.04E+02   1    3.36E-04
  42.0   7.24E+02   3.458E+02   -7.08E+02   1    2.44E-04
  44.0   7.23E+02   3.460E+02   -7.10E+02   1    1.53E-04
  46.0   7.23E+02   3.462E+02   -7.12E+02   1    2.66E-03
  48.0   7.23E+02   3.464E+02   -7.14E+02   1    9.16E-05
  50.0   7.23E+02   3.466E+02   -7.16E+02   1    1.80E-03
  52.0   7.23E+02   3.467E+02   -7.17E+02   1    1.46E-03
Finished running program ss_thinwall
```

Figure 4.24. Listing of output file "thinwall_out" produced by the ss_thinwall code for Example Problem 3.

```
 X            I            Q            T            TCS
0.000E+00    1.000E+00     -           5.235E+02
0.000E+00    1.000E+00     -           5.367E+02
2.500E-02     -           3.319E+03     -           8.474E-01
5.000E-02    9.585E-01     -            -
1.000E-01     -           3.589E+03    5.073E+02    9.829E-01
1.500E-01    8.688E-01     -            -
1.750E-01     -           4.355E+03     -           1.294E+00
2.000E-01    8.143E-01     -           4.061E+02
Finished running program ss_neutron
```

Figure 4.25. Listing of output file "neutron0_out" produced by the ss_neutron0 code for Example Problem 3.

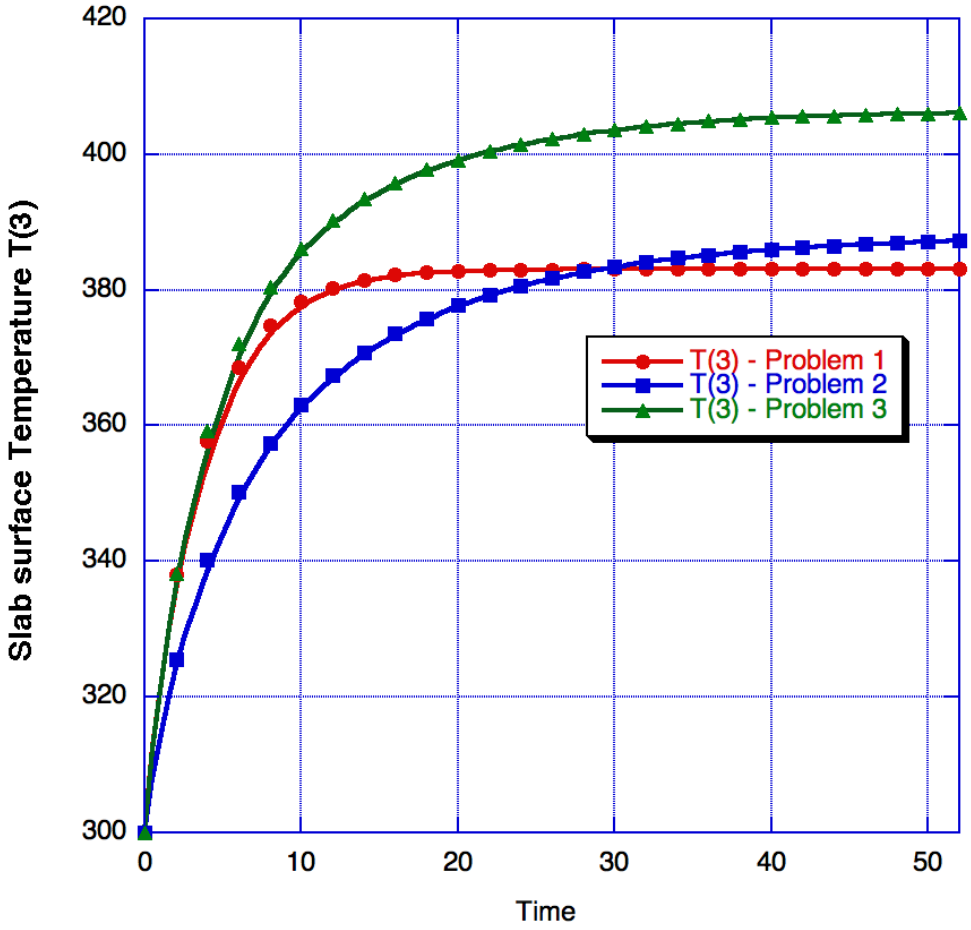


Figure 4.26. Comparison of the slab surface temperature T(3) computed in ss_con1d for Example Problems 1, 2 and 3

4.3.4 Example 4: Fixed point coupling of ss_con1d and ss_thinwall using a global residual based convergence check

In this example we couple ss_con1d and ss_thinwall using a simple fixed point approach but where the Problem Manager determines convergence by evaluating a global residual. The actual problem being solved is identical to Example 2, but in this example we see how the Problem Manager can construct a global residual vector from the local residual vectors provided by each physics code. We do this by implementing the additional interface requirements that are described in Table 2.6. For clarity, we choose to do this by creating additional Model Evaluator files which inherit from the originals used on Problem 2:

CON1D_ModelEval0_w_Resid.cpp

CON1D_ModelEval0_w_Resid.hpp

THINWALL_ModelEval0_w_Resid.cpp

THINWALL_ModelEval0_w_Resid.hpp

A complete listing of these files is given in Appendix D.3.2.

Using this approach, the only change to the Multiphysics Driver is to reference the new Model Evaluator names (i.e. *_w_Resid, compare the listing shown on Appendix D.3.1 with Appendix D.2.1).

The state variable output for this example problem is the same as already shown for Example Problem 2 (iterations counts and residual values are not identical).

4.3.5 Example 5: JFNK based coupling of ss_con1d and ss_thinwall

In this example we return to the same problem as previously solved in Example Problems 2 and 4. However, this time we wish to use the JFNK method to solve the coupled physics codes. No changes to the Multiphysics Driver used in Example Problem 4 is required, but one additional Problem_Manager/Model_Evaluator interface routine must be implemented in each of the Model Evaluators. This is an implementation of the "set_x_stat" routine. To save space, Appendix 5 lists only that portion of the Model Evaluator source (i.e. *.cpp) and header (i.e. *.hpp) files that were added to accomplish this requirement.

To run the problem, the correct LIME xml input files (Problem_Manager_setup.xml and Problem_Manager_setup_jfnk.xml) must also be present. These were previously shown in Figures 4.13 and 4.15 respectively.

The state variable output for this example problem is the same as already shown for Example Problem 2 (iterations count and residual values are not identical).

4.3.6 Example 6: JFNK based coupling of ss_con1d, ss_thinwall and ss_neutron

In this example we return to the same problem as previously solved in Example Problem 3 (which coupled each of the SuperSimple physics codes together), but this time we wish to use the JFNK method. In this case, the Multiphysics driver is almost the same, but must reference the "CON1D_ModelEval0_w_Resid" and "THINWALL_ModelEval0_w_Resid" Model Evaluators instead. Appendix D.6.1 lists the revised Multiphysics driver, and can be compared to that shown in Appendix D.3.1.

No new Model Evaluators need be written to perform Example Problem 6. This problem requires the same Model Evaluators as were used in Example Problem 5 together with the original Model Evaluator used for the ss_neutron0 code (which in total consist of 10 files).

As with Example Problem 5, to run this problem the correct LIME XML input files (Problem_Manager_setup.xml and Problem_Manager_setup_jfnk.xml) must also be present (see Figures 4.13 and 4.15 respectively).

The state variable output for this example problem is the same as already shown for Example Problem 3 (iterations count and residual values are not identical).

References

[1] Sandia National Laboratories. "NOX and LOCA, Object-Oriented Nonlinear Solver and Continuation Packages". http://www.trilinos.sandia.gov/packages/nox. 15, 20, 23

[2] Sandia National Laboratories. "The Trilinos Project". http://www.trilinos.sandia.gov. 15, 20

[3] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1983. 17, 20

[4] D. A. Knoll and D. E. Keyes. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193:357–397, 2004. 16, 17, 19

[5] H. G. Matthies and J. Steindorf. Partitioned strong coupling algorithms for fluid-structure interaction. *Computers & Structures*, 81(8-11):805–812, 2003. 17

[6] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, 1970. 17

[7] R. Pawlowski, R. A. Bartlett, N. Belcourt, R. Hooper, and R. Schmidt. A theory manual for multi-physics code coupling in LIME. Sandia Technical Report SAND2011-2195, Sandia National Laboratories, March 2011. 13, 16, 20

[8] R. C. Schmidt, K. Belcourt, K. T. Clarno, R. Hooper, L. L. Humphries, A. L. Lorber, R. J. Pryor, and W. F. Spotz. Foundational development of an advanced nuclear reactor integrated safety code. Technical Report SAND2010-0878, Sandia National Laboratories, 2010. 20

Appendix A

Original Unfactored "SuperSimple" Physics Code Listings

A.1 Original ss_con1d

```
PROGRAM ss_con1d
! ======================================================================
! Transient 1D conduction, with Heat Src and with specified surface BCs
! Spatial discretization fixed as a 3 node finite difference approximation
! Temporal discretization is first-order Euler implicit
! ======================================================================

  IMPLICIT NONE
  ! Geometry, mesh and material property related -------------------------------
  REAL    :: W      ! total width of 1-D domain
  REAL    :: dx     ! width of finite difference discretization (inner nodes)
  REAL    :: k      ! thermal conductivity
  REAL    :: rhoCp  ! density*specific heat
  ! Boundary Condition related -------------------------------------------------
  REAL    :: qlt    ! specified heat flux at the left-side boundary
  REAL    :: qrt    ! computed heat flux at the right-side boundary
  REAL    :: h_c    ! convective heat trasnfer coeficient at the left-side boundary
  REAL    :: Tf     ! fluid temperature "seen" by the left-side boundary
  REAL    :: Tr     ! far-field radiation temperature "seen" by the right-side boundary
  ! State variables and/or source term related ---------------------------------
  REAL    :: time   ! time (sec)
  REAL    :: tmax   ! maximum time
  REAL    :: dt     ! time step
  REAL    :: Qs(3)  ! heat source array
  REAL    :: T(3)   ! temperature array during solution iteration
  REAL    :: T0(3)  ! temperature array (converged) at previous time step
  ! Solver and Solution --------------------------------------------------------
  REAL    :: R(3)   ! Residual array
  REAL    :: R_max  ! maximum value in the Residual array
  REAL    :: coef1  ! coeficient used in energy equation
  REAL    :: coef2  ! coeficient used in energy equation
  REAL    :: coef3  ! coeficient used in energy equation
  REAL    :: tol    ! convergence criteria tolerance
  REAL    :: ebal   ! normalized energy balance  (=1.0 at steady state)
  REAL    :: f      ! relaxation factor for SS iteration method
  INTEGER :: n      ! time-step loop index
  INTEGER :: nstep  ! Number of time steps to take
  INTEGER :: iter   ! iteration counter during iterative solve loop
  INTEGER :: uout   ! Output unit for fortran write statements
! ======================================================================
! Problem Setup
  W    = 0.20
  dx   = W/2.0
  k    = 0.5
  rhoCp= 100.
  h_c  = 0.1        ! setting = 0 makes left side BC adiabatic
  Tf   = 350.0      ! left side fluid temperature
```

```
  Tr   = 300.0      ! right side radiation temperature
  time = 0.0
  tmax = 50.0
  dt   = 2.0
  nstep= 25         ! number of time steps set to 25
  Qs   = 3000.      ! heat source array initialized
  T    = 300.       ! Temperature array initialized
  T0   = T          !
  tol  = 2.e-3
  ebal = 0.0
  uout   = 6        ! standard output goes to unit 6
  if(uout.ne.6) OPEN(unit=uout, file='con1d_out', status='unknown')

  coef1 = dt*k/(rhoCp*dx*dx)
  coef2 = dt/(rhoCp*0.5*dx)
  coef3 = dt/rhoCp

  write(uout,'(a)') " time     qlt        T(1)        T(2)        T(3)         qrt       ebal      iter    R_max"
  write(uout,9) time, qlt,T(1),T(2),T(3),qrt,ebal,iter,R_max
9 format(f6.1,2x, es9.2, 3es12.3, 2x, 2es10.2, 2x, i4, 3x, es8.2)

! ================================================================================
! Solve problem

  DO n=1,nstep   ! Begining of the time-step loop

     ! ---------------------------------------------------------------------
     ! Solve using succesive substitution iteration method (requires relaxation factor f)
     DO iter = 1,500

        ! Update boundary heat fluxes
        qrt = 5.67e-8*(Tr**4 - T(3)**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
        qlt = h_c*(Tf - T(1))

        f = 0.20  ! I've hardwired this value here. Other conditions would require tuning
        T(1) = f*(T0(1) + 2.*coef1*(-T(1) +   T(2)        ) + coef2*qlt + coef3*Qs(1)) + (1.-f)*T(1)
        T(2) = f*(T0(2) +    coef1*( T(1) -2.*T(2) + T(3) ) +             coef3*Qs(2)) + (1.-f)*T(2)
        T(3) = f*(T0(3) + 2.*coef1*(          T(2) - T(3) ) + coef2*qrt + coef3*Qs(3)) + (1.-f)*T(3)

        ! Compute residual vector and check for convergence
        qrt =  5.67e-8*(Tr**4 - T(3)**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
        qlt = h_c*(Tf - T(1))
        R(1) = T0(1) + 2.*coef1*(-T(1) +   T(2)        ) + coef2*qlt + coef3*Qs(1) - T(1)
        R(2) = T0(2) +    coef1*( T(1) -2.*T(2) + T(3) ) +             coef3*Qs(2) - T(2)
        R(3) = T0(3) + 2.*coef1*(          T(2) - T(3) ) + coef2*qrt + coef3*Qs(3) - T(3)
        R_max = MAXVAL(ABS(R))
        ! print *, time, iter, R_max,"[",Qs(1),", ",Qs(2),", ",Qs(3),"]"  ! Debug
        if(R_max .lt. tol) EXIT   ! exit out of iteration loop when convergence criteria met

     END DO
     ! ---------------------------------------------------------------------
     IF(R_max .gt. tol) write(6,*) "Warning: Iteration did not converge!"

     T0   = T           ! f90 vector operation updating T0
     time = time + dt
     ebal = (-(qlt+qrt)/W)/(0.25*Qs(1)+0.5*Qs(2)+0.25*Qs(3))   ! must equal 1.0 at steady state
     write(6,9) time, qlt,T(1),T(2),T(3),qrt,ebal,iter,R_max

  END DO     ! End of the time-step loop

! ================================================================================
! Final output
  write(6,*) " Finished running program ss_con1d "
  if(uout.ne.6)   CLOSE(unit=uout)
! ================================================================================
END PROGRAM ss_con1d
```

A.2 Original ss_neutron

```
PROGRAM ss_neutron
! ================================================================================
! Neutron intensity as a function of distance for an extremely simplified case
! of monoenergitic neutrons where absorption is the ONLY mechanism modeled
! Code also computes the resultant energy release.
! ================================================================================

  IMPLICIT NONE
  INTEGER :: j       ! mesh index
  INTEGER :: jmax=3  ! max index value
  REAL    :: dx      ! X(j)-X(j-1)
  REAL    :: Cnq     ! Conversion factor between absorbed Neutrons and Joules
  REAL    :: X(0:3)  ! X locations within the wall (distance from X=0)
  REAL    :: I(0:3)  ! Intensity (Neutrons/unit-area) at the four X locations
  REAL    :: Q(3)    ! Energy release within wall between point X(i) and X(i-1)
  REAL    :: T(3)    ! Temperature within wall between point X(i) and X(i-1)
  REAL    :: TCS(3)  ! Total Cross Section values
  INTEGER :: uout    ! Output unit for fortran write statements

! ================================================================================
! Problem Setup
  Cnq    = 4000.
  I(0)   = 1.0      ! Neutron Intensity at X=0
  T      = 400.0

  X(0)   = 0.0      ! Mesh locations chosen to align with faces in ss_1dcon code
  X(1)   = 0.05
  X(2)   = 0.15
  X(3)   = 0.20

  write(6,*) "      index   X(j)           I(j)            Q(j)"
  write(6,*) 0, X(0), I(0)
! ================================================================================
! Solve problem

  DO j=1,3   ! Loop over the three X locations in the wall

     ! Temperature dependant Total Cross Section (I made this polynomial up)
     TCS(j) = 1.042-0.00255*T(j) + 2.36e-5*T(j)**2 - 4.08e-8*T(j)**3 + 2.86e-17*T(j)**6
     IF(T(j).gt.705.) TCS(j) = 0.189

     ! Intensity
     dx = X(j)-X(j-1)
     I(j) = I(j-1) * exp(-TCS(j)*dx)

     ! Energy Release
     Q(j) = -Cnq*(I(j)-I(j-1))/dx

  END DO

!9 format(i4, 2x, es11.3, 2x, es11.3, 2x, es11.3)
! ================================================================================
! Final output
  write(uout,*) "  X            I            Q            T            TCS"
  write(uout,7) X(0), I(0), T(1)
  DO j=1,jmax   ! Loop over the wall locations, dealing with staggered grid
     if(j.eq.1) write(uout,6) (X(j)+X(j-1))/2.,  Q(j), TCS(j)
     if(j.gt.1) write(uout,8)  X(j-1), I(j-1)
     if(j.gt.1 .and. j.lt.jmax) write(uout,9) (X(j)+X(j-1))/2.,  Q(j), T(j), TCS(j)
     if(j.gt.1 .and. j.eq.jmax) write(uout,6) (X(j)+X(j-1))/2.,  Q(j), TCS(j)
     if(j.eq.jmax) write(uout,7) X(j), I(j), T(j)
  END DO

  write(uout,*) " Finished running program ss_neutron "
```

```
  if(uout.ne.6)   CLOSE(unit=uout)

6 format( es11.3, 5x,'-',9x, es11.3, 5x,"-", 9x, es11.3)
7 format( es11.3, 2x, es11.3, 5x,"-",9x,es11.3 )
8 format( es11.3, 2x, es11.3, 5x,"-",12x,"-" )
9 format( es11.3, 5x,'-',9x, es11.3, 2x, es11.3, 2x, es11.3)
! ================================================================================
END PROGRAM ss_neutron
```

A.3 Original ss_thinwall

```
PROGRAM ss_thinwall
! ======================================================================
! Transient thermal response of a conductive thin wall subject to rad. BC on
! the left side and a convective BC on the right side.(Lumped mass approximation)
! Temporal discretization is first-order Euler implicit
! ======================================================================

  IMPLICIT NONE

  REAL    :: rhoCpW ! density*specific heat*width of wall
  REAL    :: Tfluid ! Fluid Temperature on the right-side boundary
  REAL    :: hconv  ! Convective heat trasnver coeficient on the right-side boundary
  REAL    :: Trad   ! Radiation Temperature seen by the left-side boundary
  REAL    :: time   ! time (sec)
  REAL    :: dt     ! time step
  REAL    :: T      ! temperature (lumped mass approximation) of the wall
  REAL    :: T0     ! temperature at previous time step
  REAL    :: R      ! Residual
  REAL    :: dRdT   ! derivative of Residual wrt Temperature
  REAL    :: delT   ! Newton-based temperature update
  REAL    :: coef1  ! coeficient used in energy equation
  REAL    :: coef2  ! coeficient used in energy equation
  REAL    :: coef3  ! coeficient used in energy equation
  REAL    :: tol    ! convergence criteria tolerance
  REAL    :: qrad   ! radiation heat trasnfer on the left side (per unit area)
  REAL    :: qfluid ! convective heat transfer on the right side (per unit area)
  INTEGER :: n      ! time-step loop index
  INTEGER :: iter   ! iteration counter during iterative solve loop
! ======================================================================
! Problem Setup
  rhoCpW = 100.
  Tfluid = 275.0
  hconv  = 10.
  Trad   = 400.
  time   = 0.0
  dt     = 2.0
  T      = 300.
  T0     = T
  tol    = 2.e-3
  coef1 = 5.67e-8*dt/rhoCpW  ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
  coef2 = dt*hconv/rhoCpW
  coef3 = coef1/(1.0+coef2)

  write(6,'(a)') " time     qrad        T          qfluid   iter    Resid"
  write(6,9) time, 0.0,T,0.0
! ======================================================================
! Solve problem

  DO n=1,25   ! Top of the time-step loop

     ! ---------------------------------------------------------------------
     ! Solve using a simple Newton-method-based iteration scheme
     DO iter = 1,15

        ! Compute Residual
        R = T + coef3*T**4 - (T0 + coef1*Trad**4 + coef2*Tfluid)/(1.0+coef2)

        ! Check for convergence
        if(abs(R) .lt. tol) EXIT   ! exit out of iteration loop when convergence criteria met

        ! Derivative of the Residual with respect to T ( 1x1 jacobian)
        dRdT = 1.0 + 4.0*coef3*T**3

        ! Newton update
```

```
        delT = -R / dRdT

        ! write(6,'(a,i3,a,3es11.3)') "DEBUG: iter=", iter, "  Res, delT, T+delT= ",R, delT, T+delT

        ! update to temperature estimate
        T = T + delT
      END DO
      ! ------------------------------------------------------------------------
      IF(abs(R) .gt. tol) write(6,*) "Warning: ThinWall Iteration did not converge!"

      T0   = T
      time = time + dt
      qrad =  5.67e-8*(Trad**4 - T**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
      qfluid = hconv*(Tfluid-T)
      write(6,9) time, qrad,T,qfluid,iter, abs(R)

  END DO     ! Bottom of the time-step loop

9 format(f6.1,2x, es9.2, es12.3, 2x, es10.2, i4, 3x, es8.2)
! ============================================================================
! Final output
  write(6,*) " Finished running program ss_thinwall "
! ============================================================================
END PROGRAM ss_thinwall
```

Appendix B

Refactored "SuperSimple" Physics Code Listings

B.1 Revised ss_con1d

```
!*******************************************************************************************
! Transient 1D conduction, with Heat Src and with specified surface BCs
! Spatial discretization fixed as a 3 node finite difference approximation
! Temporal discretization is fully implicit first-order Euler
!
! This file contains the following program units
! MODULE     con1d_mod
! PROGRAM    ss_con1d
! SUBROUTINE setup_con1d
! SUBROUTINE solve_con1d
! SUBROUTINE finish_con1d
! SUBROUTINE take_time_step_con1d
! SUBROUTINE s_substitution_con1d
! SUBROUTINE update_con1d
! SUBROUTINE residual_con1d
! SUBROUTINE jacobian_con1d  (not currently used, has not been tested and debugged)
! *******************************************************************************************


!*******************************************************************************************
MODULE con1d_mod
!*******************************************************************************************
  IMPLICIT NONE
  PUBLIC

  ! Geometry, mesh and material property related -------------------------------
  REAL    :: W      ! total width of 1-D domain
  REAL    :: dx     ! width of finite difference discretization
  REAL    :: k      ! thermal conductivity
  REAL    :: rhoCp  ! density*specific heat
  ! Boundary Condition related -------------------------------------------------
  REAL    :: qlt    ! specified heat flux at the left-side boundary
  REAL    :: qrt    ! computed heat flux at the right-side boundary
  REAL    :: h_c    ! convective heat trasnfer coeficient at the left-side boundary
  REAL    :: Tf     ! fluid temperature "seen" by the left-side boundary
  REAL    :: Tr     ! far-field radiation temperature "seen" by the right-side boundary
  ! State variables and/or source term related ---------------------------------
  REAL    :: time   ! time (sec)
  REAL    :: tmax   ! maximum time
  REAL    :: dt     ! time step
  REAL    :: Qs(3)  ! heat source array
  REAL    :: T(3)   ! temperature array during solution iteration
  REAL    :: T0(3)  ! temperature array (converged) at previous time step
  ! Solver and Solution --------------------------------------------------------
```

```
  REAL    :: R(3)   ! Residual array
  REAL    :: R_max  ! maximum value in the Residual array
  REAL    :: coef1  ! coeficient used in energy equation
  REAL    :: coef2  ! coeficient used in energy equation
  REAL    :: coef3  ! coeficient used in energy equation
  REAL    :: tol    ! convergence criteria tolerance
  REAL    :: ebal   ! normalized energy balance  (=1.0 at steady state)
  REAL    :: f      ! relaxation factor for SS iteration method
  INTEGER :: iter   ! iteration counter during iterative solve loop
  INTEGER :: uout   ! Output unit for fortran write statements
  INTEGER :: n_vars ! Number of unknowns and residual equations

END MODULE con1d_mod
! ========================

!*******************************************************************************************
!PROGRAM ss_con1d
!*******************************************************************************************
!  IMPLICIT NONE
!
! Problem Setup
!  CALL setup_con1d
!
! Solve problem
!  CALL solve_con1d(25)
!
! Final output
!  CALL finish_con1d
!
!END PROGRAM ss_con1d
! ========================

!*******************************************************************************************
SUBROUTINE setup_con1d
!*******************************************************************************************
  USE con1d_mod
  IMPLICIT NONE

  W    = 0.20
  dx   = W/2.0
  k    = 0.5
  rhoCp= 100.
  h_c  = 0.1        ! setting = 0 makes left side BC adiabatic
  Tf   = 350.0      ! left side fluid temperature
  Tr   = 300.0      ! right side radiation temperature
  time = 0.0
  tmax = 50.0
  dt   = 2.0
  Qs   = 3000.      ! heat source array initialized
  T    = 300.       ! Temperature array initialized
  T0   = T          !
  tol  = 2.e-3      !
  ebal = 0.0
  uout   = 8        ! standard output goes to unit 6
  n_vars = 3        ! # unknowns (variables) and corresponding residual equations
  if(uout.ne.6) OPEN(unit=uout, file='con1d_out', status='unknown')

  coef1 = dt*k/(rhoCp*dx*dx)
  coef2 = dt/(rhoCp*0.5*dx)
  coef3 = dt/rhoCp

  write(uout,'(a)') " time     qlt        T(1)        T(2)        T(3)         qrt       ebal      iter    R_max"
  write(uout,9) time, qlt,T(1),T(2),T(3),qrt,ebal,iter,R_max
9 format(f8.1,2x, es9.2, 3es12.3, 2x, 2es10.2, 2x, i4, 3x, es9.2)

END SUBROUTINE setup_con1d
! ========================
```

```
!*****************************************************************************************
SUBROUTINE solve_con1d(nstep)
!*****************************************************************************************
  IMPLICIT NONE
  INTEGER :: n, nstep

  DO n=1,nstep   ! Begining of the time-step loop

     CALL take_time_step_con1d

     CALL update_con1d

  END DO     ! End of the time-step loop

END SUBROUTINE solve_con1d
! ========================


!*****************************************************************************************
SUBROUTINE finish_con1d
!*****************************************************************************************
  USE con1d_mod
  IMPLICIT NONE

  write(uout,*) " Finished running program ss_con1d "
  if(uout.ne.6)   CLOSE(unit=uout)

END SUBROUTINE finish_con1d
! ========================


!*****************************************************************************************
SUBROUTINE take_time_step_con1d
!*****************************************************************************************
  IMPLICIT NONE

  INTEGER :: METHOD   !Solver method flag: 1 = Successive substitution, else Newtons Method
  METHOD = 1

!  IF(METHOD .eq. 1) THEN

     CALL s_substitution_con1d

!  ELSE
!     CALL newton_con1d
!  ENDIF

END SUBROUTINE take_time_step_con1d
! ========================

!*****************************************************************************************
SUBROUTINE s_substitution_con1d
!*****************************************************************************************
  USE con1d_mod
  IMPLICIT NONE
  INTEGER i

     ! ------------------------------------------------------------------
     ! Solve using succesive substitution iteration method (requires relaxation factor f)
     DO i = 1,500

        ! Update boundary heat fluxes
        qrt = 5.67e-8*(Tr**4 - T(3)**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
        qlt = h_c*(Tf - T(1))

        f = 0.20  ! I've hardwired this value here. Other conditions would require tuning
        T(1) = f*(T0(1) + 2.*coef1*(-T(1) +   T(2)        ) + coef2*qlt + coef3*Qs(1)) + (1.-f)*T(1)
        T(2) = f*(T0(2) +    coef1*( T(1) -2.*T(2) + T(3) ) +             coef3*Qs(2)) + (1.-f)*T(2)
```

```
        T(3) = f*(T0(3) + 2.*coef1*(          T(2) - T(3) ) + coef2*qrt + coef3*Qs(3)) + (1.-f)*T(3)

        ! Compute residual vector and check for convergence
        CALL residual_con1d
        if(R_max .lt. tol) THEN   ! exit out of iteration loop when convergence criteria met
          ! write(uout,*) "CON1D Iteration converged in ",i," iterations."
          iter = iter + i
          EXIT
        endif

     END DO
     ! ------------------------------------------------------------------------
     IF(R_max .gt. tol) write(uout,*) "Warning: CON1D Iteration did not converge (",R_max," > ",tol,") !"

END SUBROUTINE s_substitution_con1d
! ========================

!*******************************************************************************************
SUBROUTINE update_con1d
!*******************************************************************************************
  USE con1d_mod
  IMPLICIT NONE
  REAL :: Qbar  ! volume weighted average heat source

  T0   = T           ! f90 vector operation updating T0
  time = time + dt
  Qbar =  0.25*Qs(1)+0.5*Qs(2)+0.25*Qs(3)
  ebal = (-(qlt+qrt)/W)/Qbar   ! must equal 1.0 at steady state
  write(uout,9) time, qlt,T(1),T(2),T(3),qrt,ebal,iter,R_max

  ! Reset iter counter
  iter = 0

9 format(f6.1,2x, es9.2, 3es12.3, 2x, 2es10.2, 3x, i4, 2x, es8.2)

END SUBROUTINE update_con1d
! ========================


!*******************************************************************************************
SUBROUTINE residual_con1d
!*******************************************************************************************
  USE con1d_mod
  IMPLICIT NONE

  ! Compute residual vector and check for convergence
  qrt =  5.67e-8*(Tr**4 - T(3)**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
  qlt = h_c*(Tf - T(1))
  R(1) = T0(1) + 2.*coef1*(-T(1) +   T(2)        ) + coef2*qlt + coef3*Qs(1) - T(1)
  R(2) = T0(2) +    coef1*( T(1) -2.*T(2) + T(3) ) +             coef3*Qs(2) - T(2)
  R(3) = T0(3) + 2.*coef1*(          T(2) - T(3) ) + coef2*qrt + coef3*Qs(3) - T(3)
  R_max = MAXVAL(ABS(R))
  ! print *, time, iter, R_max,"[",Qs(1),", ",Qs(2),", ",Qs(3),"]"  ! Debug

END SUBROUTINE residual_con1d
! ========================

!*******************************************************************************************
SUBROUTINE jacobian_con1d
!*******************************************************************************************
  USE con1d_mod
  IMPLICIT NONE

  REAL :: J(3,3)  ! Jacobian matrix.  Note  J(eq_index, T_index)

  ! Compute Jacobian matrix
  J(1,1) = -(1+ 2.*coef1) - coef2*h_c
  J(1,2) =   2.*coef1
```

```
  J(1,3) =   0.0
  J(2,1) =   coef1
  J(2,2) = -(1+ 2.*coef1)
  J(2,3) =   coef1
  J(3,1) =   0.0
  J(3,2) =   2.*coef1
  J(3,3) = -(1+ 2.*coef1) - coef2*5.67e-8*4.0*T(3)**3   ! This is the only non-linear term

END SUBROUTINE jacobian_con1d
! ========================
```

B.2 Revised ss_neutron

```
!*******************************************************************************************
! Neutron intensity as a function of distance for an extremely simplified case
! of monoenergetic neutrons where absorption is the ONLY mechanism modeled
! Code also computes the resultant energy release.
!
! This file contains the following program units
! MODULE     neutron0_mod
! PROGRAM    ss_neutron0
! SUBROUTINE setup_neutron0
! SUBROUTINE solve_neutron0
! SUBROUTINE finish_neutron0
!*******************************************************************************************


!*******************************************************************************************
MODULE neutron0_mod
!*******************************************************************************************
  IMPLICIT NONE
  PUBLIC

  REAL    :: Cnq      ! Conversion factor between absorbed Neutrons and Joules
  REAL    :: X(0:3)   ! X locations within the wall (distance from X=0)
  REAL    :: I(0:3)   ! Intensity (Neutrons/unit-area) at the four X locations
  REAL    :: Q(3)     ! Energy release within wall between point X(i) and X(i-1)
  REAL    :: T(3)     ! Temperature within wall between point X(i) and X(i-1)
  REAL    :: RHS(3)   ! Right-hand-side for assignment of Q values
  REAL    :: TCS(3)   ! Total Cross Section values
  REAL    :: R(3)     ! Residual array
  REAL    :: R_max    ! maximum value in the Residual array
  INTEGER :: uout     ! Output unit for fortran write statements

END MODULE neutron0_mod
! ========================


!*******************************************************************************************
PROGRAM ss_neutron0
!*******************************************************************************************
!  IMPLICIT NONE
!
! Problem Setup
!  CALL setup_neutron0
!
! Solve problem
!  CALL solve_neutron0
!
! Final output
!  CALL finish_neutron0
!
END PROGRAM ss_neutron0
```

```
! ========================


!*****************************************************************************************
SUBROUTINE setup_neutron0
!*****************************************************************************************
  USE neutron0_mod
  IMPLICIT NONE

! Problem Setup
  X(0)   = 0.0      ! Mesh locations chosen to align with faces in ss_1dcon code
  X(1)   = 0.05
  X(2)   = 0.15
  X(3)   = 0.20

  Cnq    = 4000.
  I(0)   = 1.0      ! Neutron Intensity at X=0
  T      = 400.0    ! Temperature array set to 400.0
  CALL solve_neutron0 ! initialize q with initial T
  uout   = 9        ! standard output goes to unit 6
  if(uout.ne.6)   OPEN(unit=uout, file='neutron0_out', status='unknown')

END SUBROUTINE setup_neutron0
! ========================


!*****************************************************************************************
SUBROUTINE solve_neutron0
!*****************************************************************************************
  USE neutron0_mod
  IMPLICIT NONE
  REAL    :: dx      ! X(j)-X(j-1)
  INTEGER :: j       ! index

  ! Solve problem
  DO j=1,3   ! Loop over the three X locations in the wall

     ! Temperature dependant Total Cross Section (I made this polynomial up)
     TCS(j) = 1.042-0.00255*T(j) + 2.36e-5*T(j)**2 - 4.08e-8*T(j)**3 + 2.86e-17*T(j)**6
     IF(T(j).gt.705.) TCS(j) = 0.189

     ! Intensity
     dx = X(j)-X(j-1)
     I(j) = I(j-1) * exp(-TCS(j)*dx)

     ! Energy Release
      Q(j) = -Cnq*(I(j)-I(j-1))/dx

  END DO

END SUBROUTINE solve_neutron0
! ========================


!*****************************************************************************************
SUBROUTINE finish_neutron0
!*****************************************************************************************
  USE neutron0_mod
  IMPLICIT NONE
  INTEGER :: j, jmax=3

! Final output
  write(uout,*) "  X            I            Q            T            TCS"
  write(uout,7) X(0), I(0), T(1)
  DO j=1,jmax   ! Loop over the wall locations, dealing with staggered grid
     if(j.eq.1) write(uout,6) (X(j)+X(j-1))/2.,  Q(j), TCS(j)
     if(j.gt.1) write(uout,8)  X(j-1), I(j-1)
     if(j.gt.1 .and. j.lt.jmax) write(uout,9) (X(j)+X(j-1))/2.,  Q(j), T(j), TCS(j)
```

```
     if(j.gt.1 .and. j.eq.jmax) write(uout,6) (X(j)+X(j-1))/2.,  Q(j), TCS(j)
     if(j.eq.jmax) write(uout,7) X(j), I(j), T(j)
  END DO

  write(uout,*) " Finished running program ss_neutron "
  if(uout.ne.6)   CLOSE(unit=uout)

6 format( es11.3, 5x,'-',9x, es11.3, 5x,"-", 9x, es11.3)
7 format( es11.3, 2x, es11.3, 5x,"-",9x,es11.3 )
8 format( es11.3, 2x, es11.3, 5x,"-",12x,"-" )
9 format( es11.3, 5x,'-',9x, es11.3, 2x, es11.3, 2x, es11.3)

END SUBROUTINE finish_neutron0
! ========================
```

B.3 Revised ss_thinwall

```
!*****************************************************************************************
! Transient thermal response of a conductive thin wall subject to rad. BC on
! the left side and a convective BC on the right side.(Lumped mass approximation)
! Temporal discretization is first-order Euler implicit
!
! This file contains the following program units
! MODULE     thinwall_mod
! PROGRAM    ss_thinwall
! SUBROUTINE setup_thinwall
! SUBROUTINE solve_thinwall
! SUBROUTINE finish_thinwall
! SUBROUTINE timestep_thinwall
! SUBROUTINE update_thinwall
! SUBROUTINE residual_thinwall
! SUBROUTINE residual2_thinwall  (thread safe version)
! *****************************************************************************************


!*****************************************************************************************
MODULE thinwall_mod
!*****************************************************************************************
  IMPLICIT NONE
  PUBLIC

  REAL    :: rhoCpW ! density*specific heat*width of wall
  REAL    :: Tfluid ! Fluid Temperature on the right-side boundary
  REAL    :: hconv  ! Convective heat trasnver coeficient on the right-side boundary
  REAL    :: Trad   ! Radiation Temperature seen by the left-side boundary
  REAL    :: time   ! time (sec)
  REAL    :: dt     ! time step
  REAL    :: T(1)   ! temperature (lumped mass approximation) of the wall
  REAL    :: T0     ! temperature at previous time step
  REAL    :: R(1)   ! Residual array
  REAL    :: dRdT   ! derivative of Residual wrt Temperature
  REAL    :: delT   ! Newton-based temperature update
  REAL    :: coef1  ! coeficient used in energy equation
  REAL    :: coef2  ! coeficient used in energy equation
  REAL    :: coef3  ! coeficient used in energy equation
  REAL    :: tol    ! convergence criteria tolerance
  REAL    :: qrad   ! radiation heat trasnfer on the left side (per unit area)
  REAL    :: qfluid ! convective heat transfer on the right side (per unit area)
  INTEGER :: n      ! time-step loop index
  INTEGER :: iter   ! iteration counter during iterative solve loop
  INTEGER :: uout   ! Output unit for fortran write statements
  INTEGER :: n_vars ! Number of unknowns and residual equations

END MODULE thinwall_mod
```

```
! ========================


!*****************************************************************************************
!PROGRAM ss_thinwall
!*****************************************************************************************
!  IMPLICIT NONE
!
! Problem Setup
!  CALL setup_thinwall
!
! Solve problem
!  CALL solve_thinwall(25)
!
! Final output
!  CALL finish_thinwall
!
!END PROGRAM ss_thinwall
! ========================



!*****************************************************************************************
SUBROUTINE setup_thinwall
!*****************************************************************************************
  USE thinwall_mod
  IMPLICIT NONE

  rhoCpW = 100.
  Tfluid = 275.0
  hconv  = 10.
  Trad   = 400.
  time   = 0.0
  dt     = 2.0
  T(1)   = 300.
  T0     = T(1)
  tol    = 2.e-3

  coef1 = 5.67e-8*dt/rhoCpW  ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
  coef2 = dt*hconv/rhoCpW
  coef3 = coef1/(1.0+coef2)

  n_vars = 1        ! # unknowns (variables) and corresponding residual equations
  uout   = 11       ! standard output goes to unit 11
  if(uout.ne.6)   OPEN(unit=uout, file='thinwall_out', status='unknown')
  write(uout,'(a)') " time     qrad        T          qfluid   iter    Resid"
  write(uout,9) time, 0.0,T(1),0.0
9 format(f8.1,2x, es9.2, es12.3, 2x, es10.2, i4, 3x, es9.2)

END SUBROUTINE setup_thinwall
! ========================


!*****************************************************************************************
SUBROUTINE solve_thinwall(nstep)
!*****************************************************************************************
  IMPLICIT NONE
  INTEGER :: n, nstep

  DO n=1,nstep   ! Begining of the time-step loop

     CALL time_step_thinwall

     CALL update_thinwall

  END DO     ! End of the time-step loop
```

```
END SUBROUTINE solve_thinwall
! ========================


!*******************************************************************************************
SUBROUTINE finish_thinwall
!*******************************************************************************************
  USE thinwall_mod, only: uout
  IMPLICIT NONE

  write(uout,*) " Finished running program ss_thinwall "

END SUBROUTINE finish_thinwall
! ========================


!*******************************************************************************************
SUBROUTINE time_step_thinwall
!*******************************************************************************************
  USE thinwall_mod
  IMPLICIT NONE

     ! ------------------------------------------------------------------------
     ! Solve using a simple Newton-method-based iteration scheme
     DO iter = 1,15

        ! Compute Residual
        ! CALL residual_thinwall
        CALL residual2_thinwall(T(1),T0,Trad,Tfluid,coef1,coef2,coef3, R)
        ! Check for convergence
        if(abs(R(1)) .lt. tol) EXIT   ! exit out of iteration loop when convergence criteria met

        ! Derivative of the Residual with respect to T ( 1x1 jacobian)
        dRdT = 1.0 + 4.0*coef3*T(1)**3

        ! Newton update
        delT = -R(1) / dRdT

        ! write(uout,'(a,i3,a,3es11.3)') "DEBUG: iter=", iter, "  Res, delT, T+delT= ",R(1), delT, T+delT

        ! update to temperature estimate
        T(1) = T(1) + delT
     END DO
     ! ------------------------------------------------------------------------
     IF(abs(R(1)) .gt. tol) write(uout,*) "Warning: ThinWall Iteration did not converge!"


END SUBROUTINE time_step_thinwall
! ========================


!*******************************************************************************************
SUBROUTINE update_thinwall
!*******************************************************************************************
  USE thinwall_mod
  IMPLICIT NONE

     T0   = T(1)
     time = time + dt
     qrad =  5.67e-8*(Trad**4 - T(1)**4)    ! Stefan-Boltzmann constant = 5.57e-8 W/(m^2 K^4)
     qfluid = hconv*(Tfluid-T(1))
     write(uout,9) time, qrad,T(1),qfluid,iter, abs(R)

9 format(f8.1,2x, es9.2, es12.3, 2x, es10.2, i4, 3x, es9.2)

END SUBROUTINE update_thinwall
! ========================
```

```
!*******************************************************************************************
SUBROUTINE residual_thinwall
!*******************************************************************************************
  USE thinwall_mod
  IMPLICIT NONE

  R(1) = T(1) + coef3*T(1)**4 - (T0 + coef1*Trad**4 + coef2*Tfluid)/(1.0+coef2)

END SUBROUTINE residual_thinwall
! ========================



!*******************************************************************************************
SUBROUTINE residual2_thinwall(T,T0,Trad,Tfluid,coef1,coef2,coef3, R)
!*******************************************************************************************
  IMPLICIT NONE
  REAL,INTENT(in)  :: T      ! temperature (lumped mass approximation) of the wall
  REAL,INTENT(in)  :: T0     ! temperature at previous time step
  REAL,INTENT(in)  :: Trad   ! Radiation Temperature seen by the left-side boundary
  REAL,INTENT(in)  :: Tfluid ! Fluid Temperature on the right-side boundary
  REAL,INTENT(in)  :: coef1  ! coeficient used in energy equation
  REAL,INTENT(in)  :: coef2  ! coeficient used in energy equation
  REAL,INTENT(in)  :: coef3  ! coeficient used in energy equation
  REAL,INTENT(out) :: R(1)   ! Residual
! -------------------------------------------------------------------

  R(1) = T + coef3*T**4 - (T0 + coef1*Trad**4 + coef2*Tfluid)/(1.0+coef2)

END SUBROUTINE residual2_thinwall
! ========================
```

Appendix C

Standalone Multiphysics Driver and Model Evaluator Listings

C.1 Simple Multiphysics Driver for Standalone ss_con1d

```
// ************************************************************************************************
//
// con1d_sad.cpp
// This is the LIME stand-alone driver for the super-simple con1d code
//
// ************************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d physics
#include "CON1D_ModelEval.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ------------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
```

```
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
          LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create a pointer to an instance of a "CON1D_ModelEval" object called "con1d"
    //    (on the heap)
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

          Teuchos::RCP<CON1D_ModelEval> con1d =
                   Teuchos::rcp(new CON1D_ModelEval(pm,"CON1DModelEval"));


    // 4  Register or "add" our physics problem "con1d", with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "con1d_id" from the Problem Manager.

           int con1d_id = pm.add_problem(con1d);
           pm.register_complete(); // Trigger setup of groups, solvers, etc.
           pm.output_status(cout);

    // 5  We are now ready to let the problem manager drive the coupled problem

           pm.integrate();

  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
             rc = -2;
  }

  return rc;
}
```

C.2 Simple Model Evaluator for Standalone ss_con1d

C.2.1 Source code listing

```
// ************************************************************************************
// CON1D_ModelEval.cpp
// This is a LIME model evaluator for a standalone wrap of the ss_con1d.f90 code
// ************************************************************************************
//
#include "CON1D_ModelEval.hpp"
```

```
#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Problem_Manager;

// Data from fortran routines are referenced using an extern C block as
// shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

 extern "C" {
   #define setup_con1d  LIME_MANGLING_GLOBAL(setup_con1d, SETUP_CON1D)
   #define solve_con1d  LIME_MANGLING_GLOBAL(solve_con1d, SOLVE_CON1D)
   #define finish_con1d LIME_MANGLING_GLOBAL(finish_con1d, FINISH_CON1D)

   void setup_con1d();
   void solve_con1d(int*);
   void finish_con1d();
 }

//------------CON1D_ModelEval C++ constructor ----------------------------------------

CON1D_ModelEval::CON1D_ModelEval(LIME::Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name)
{
  setup_con1d();           // This calls the con1d code's setup_con1d routine
}

//------------CON1D_ModelEval solve_standalone ---------------------------------------

void CON1D_ModelEval::solve_standalone()
{
    int nstep=25;
    solve_con1d(&nstep);   // This calls the con1d code's solve_con1d routine
}

//------------CON1D_ModelEval C++ destructor -----------------------------------------

CON1D_ModelEval::~CON1D_ModelEval()
{
   finish_con1d();         // This calls the con1d code's finish_con1d routine
}
```

C.2.2 Header file listing

```
// ********************************************************************************
// CON1D_ModelEval.hpp
// This is a LIME model evaluator header file for a standalone wrap of the ss_con1d.f90 code
// ********************************************************************************
//
#ifndef LIME_EXAMPLE_CON1D_MODELEVAL_HPP
#define LIME_EXAMPLE_CON1D_MODELEVAL_HPP

#include "LIME_Model_Evaluator.hpp"

class CON1D_ModelEval : public LIME::Model_Evaluator
{

  public:

    CON1D_ModelEval(LIME::Problem_Manager & pm, const string & name);   // C++ constructor

    virtual ~CON1D_ModelEval();                                         // C++ destructor

    // Tells the LIME Problem Manager that this code supports a stand-alone solve
```

```
    virtual bool supports_standalone_solve() const { return true; }

    // The actual implementation of "solve_standalone" must be defined in the c++ source file
    virtual void solve_standalone();

};
#endif
```

C.3 Simple Multiphysics Driver for Standalone ss_neutron

```
// ********************************************************************************************
//
// neutron0_sad.cpp
// This is a LIME stand-alone driver for the ss_neutron0.f90 code
//
// ********************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// neutron physics
#include "NEUTRON_ModelEval0.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ---------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
```

```
        LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create a pointer to an instance of a "NEUTRON_ModelEval0" object called "neutron"
    //    (on the heap)
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

        Teuchos::RCP<NEUTRON_ModelEval0> neutron =
                Teuchos::rcp(new NEUTRON_ModelEval0(pm,"NEUTRONModelEval"));


    // 4  Register or "add" our physics problem "neutron", with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "neutron_id" from the Problem Manager.

         int neutron_id = pm.add_problem(neutron);
         pm.register_complete(); // Trigger setup of groups, solvers, etc.
         pm.output_status(cout);

    // 5  We are now ready to let the problem manager drive the coupled problem

         pm.integrate();

  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
          rc = -2;
  }

  return rc;
}
```

C.4 Simple Model Evaluator for Standalone ss_neutron

C.4.1 Source file listing

```
// ************************************************************************************************
// NEUTRON_ModelEval0.cpp
// This is a LIME model evaluator for a standalone wrap of ss_neutron0.f90 code
// ************************************************************************************************
//
#include "NEUTRON_ModelEval0.hpp"
#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Problem_Manager;

// Data from fortran routines are referenced using an extern C block as
// shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h
```

```
extern "C" {
  #define setup_neutron0  LIME_MANGLING_GLOBAL(setup_neutron0, SETUP_NEUTRON0)
  #define solve_neutron0  LIME_MANGLING_GLOBAL(solve_neutron0, SOLVE_NEUTRON0)
  #define finish_neutron0 LIME_MANGLING_GLOBAL(finish_neutron0, FINISH_NEUTRON0)

  void setup_neutron0();
  void solve_neutron0();
  void finish_neutron0();
}

//------------NEUTRON_ModelEval0 C++ constructor -----------------------------------------

NEUTRON_ModelEval0::NEUTRON_ModelEval0(Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name)
{
  setup_neutron0();
}

//-------------NEUTRON_ModelEval0 solve_standalone ----------------------------------------

void NEUTRON_ModelEval0::solve_standalone()
{
    solve_neutron0();
}

//------------NEUTRON_ModelEval0 C++ destructor -----------------------------------------

NEUTRON_ModelEval0::~NEUTRON_ModelEval0()
{
   finish_neutron0();
}
```

C.4.2 Header file listing

```
// ***********************************************************************************************
//
// NEUTRON_ModelEval0.hpp
// This is a LIME model evaluator header file for a standalone wrap of the ss_neutron0.f90 code
//
// ***********************************************************************************************
#ifndef LIME_EXAMPLE_NEUTRON_MODELEVAL0_HPP
#define LIME_EXAMPLE_NEUTRON_MODELEVAL0_HPP

#include "LIME_Model_Evaluator.hpp"

class NEUTRON_ModelEval0 : public LIME::Model_Evaluator
{

  public:

    NEUTRON_ModelEval0(LIME::Problem_Manager & pm, const string & name);    // C++ constructor

    virtual ~NEUTRON_ModelEval0();                                          // C++ destructor

    // Tells the LIME Problem Manager that this code supports a stand-alone solve
    virtual bool supports_standalone_solve() const { return true; }

    // The actual implementation of "solve_standalone" must be defined in the c++ source file
    virtual void solve_standalone();

};
#endif
```

C.5 Simple Multiphysics Driver for Standalone ss_thinwall

```
// ********************************************************************************************
//
// thinwall_sad.cpp
// This is the LIME stand-alone driver for the super-simple thinwall code
//
// ********************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// thinwall physics
#include "THINWALL_ModelEval.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ---------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
          LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create a pointer to an instance of a "THINWALL_ModelEval" object called "thinwall"
    //    (on the heap)
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

          Teuchos::RCP<THINWALL_ModelEval> thinwall =
                   Teuchos::rcp(new THINWALL_ModelEval(pm,"THINWALLModelEval"));
```

```
    // 4  Register or "add" our physics problem "thinwall", with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "thinwall_id" from the Problem Manager.

            int thinwall_id = pm.add_problem(thinwall);
            pm.register_complete(); // Trigger setup of groups, solvers, etc.
            pm.output_status(cout);

    // 5  We are now ready to let the problem manager drive the coupled problem

            pm.integrate();

  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
             rc = -2;
  }

  return rc;
}
```

C.6 Simple Model Evaluator for Standalone ss_thinwall

C.6.1 Source file listing

```
// ************************************************************************************
// THINWALL_ModelEval.cpp
// This is a LIME model evaluator for a standalone wrap of the super-simple thinwall code
// ************************************************************************************
//
#include "THINWALL_ModelEval.hpp"
#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Problem_Manager;

// Data from fortran routines are referenced using an extern C block as
// shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

 extern "C" {
   #define setup_thinwall     LIME_MANGLING_GLOBAL(setup_thinwall, SETUP_THINWALL)
   #define finish_thinwall    LIME_MANGLING_GLOBAL(finish_thinwall, FINISH_THINWALL)
   #define solve_thinwall     LIME_MANGLING_GLOBAL(solve_thinwall, SOLVE_THINWALL)

   void setup_thinwall();
   void finish_thinwall();
   void solve_thinwall(int*);
 }

//------------THINWALL ModelEval C++ constructor  -----------------------------------------
```

```
THINWALL_ModelEval::THINWALL_ModelEval(Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name)
{
  setup_thinwall();
}

//------------THINWALL ModelEval C++ destructor -----------------------------------------

THINWALL_ModelEval::~THINWALL_ModelEval()
{
   finish_thinwall();
}

//------------THINWALL_ModelEval Implimentation of solve_standalone -------------------------

void THINWALL_ModelEval::solve_standalone()
{
   int nstep=25;
   solve_thinwall_(&nstep);
}
```

C.6.2 Header file listing

```
// ***************************************************************************************
//
// THINWALL_ModelEval.hpp
// This is a LIME model evaluator header file for a standalone wrap of the super-simple thinwall code
// ***************************************************************************************
//
#ifndef LIME_EXAMPLE_THINWALL_MODELEVAL_HPP
#define LIME_EXAMPLE_THINWALL_MODELEVAL_HPP

#include "LIME_Model_Evaluator.hpp"

class THINWALL_ModelEval : public LIME::Model_Evaluator
{

  public:

    THINWALL_ModelEval(LIME::Problem_Manager & pm, const string & name);   // C++ constructor

    virtual ~THINWALL_ModelEval();                                         // C++ destructor

    // Tells the LIME problem manger that this code supports a stand-alone solve
    virtual bool supports_standalone_solve() const { return true; }

    // The actual implementation of "solve_standalone" must be defined in the c++ source file
    virtual void solve_standalone();

};
#endif
```

Appendix D

Multiphysics Driver and Model Evaluator File Listings for Example Applications

D.1 Example 1. Fixed-Point Coupling of ss_con1d and ss_neutron using non-linear elimination

D.1.1 Multiphysics Driver for Example 1

```
// ***********************************************************************************************
//
// lime_mpd0.cpp
// This is a LIME driver for coupling the "super-simple" codes ss_con1d and ss_neutron0
// using the following Model Evaluators: CON1D_ModelEval0, and NEUTRON_ModelEval0
//
// ***********************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d and neutron physics
#include "CON1D_ModelEval0.hpp"
#include "NEUTRON_ModelEval0.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ---------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {
```

```
int rc = 0;  // return code will be set to a non zero value on errors

try {    // the "try" construct is used to allow for catching "exceptions" that might
         // unexpectedly occur.  They can occur at any point in the program's
         // call stack so our exception handling policy is to catch all thrown excptions
         // here in main, output a diagnostic message, and terminate the program cleanly.


  // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
  //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
  //    must be constructed with either an MPI communicator or, if we are not using MPI, a
  //    class that has the same interface as the MPI communicator but does nothing.

        Epetra_Comm* comm = new Epetra_SerialComm;

  // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
  //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

        bool verbose = false;
        LIME::Problem_Manager pm(*comm, verbose);


  // 3  Create pointers to an instance of a "CON1D_ModelEval0" object called "con1d"
  //    and an instance of a "NEUTRON_ModelEval0" object called "neutron"
  //    (on the heap)
  //    Note: There are some Trilinos-specific implementation details here. For example,
  //          Teuchos is a Trilinos library and RCP is a reference counted pointer

        Teuchos::RCP<CON1D_ModelEval0> con1d =
                 Teuchos::rcp(new CON1D_ModelEval0(pm,"CON1DModelEval"));

        Teuchos::RCP<NEUTRON_ModelEval0> neutron =
                 Teuchos::rcp(new NEUTRON_ModelEval0(pm,"NEUTRONModelEval"));

  // 4  Register or "add" our physics problems "con1d" and "neutron", with
  //    the Problem manager. When doing this, we get back values for the
  //    identifier "con1d_id" and "neutron_id" from the Problem Manager.

        pm.add_problem(con1d);
        pm.add_problem(neutron);

  // 5  Create and setup two LIME data transfer operators that we will need

        LIME::Data_Transfer_Operator* p_n2c = new neutronics_2_conduction(neutron, con1d);
        LIME::Data_Transfer_Operator* p_c2n = new conduction_2_neutronics(con1d, neutron);

        Teuchos::RCP<LIME::Data_Transfer_Operator> n2c_op = Teuchos::rcp(p_n2c);
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2n_op = Teuchos::rcp(p_c2n);


  // 6  Register or "add" our two transfer operators "c2n" and "n2c" with
  //    the Problem manager

        pm.add_preelimination_transfer(c2n_op);
        pm.add_postelimination_transfer(n2c_op);

        pm.register_complete(); // Trigger setup of groups, solvers, etc.
        pm.output_status(cout);


  // 7  We are now ready to let the problem manager drive the coupled problem

         pm.integrate();
}
catch (exception& e)
{
  cout << e.what() << endl
```

```
        << "\n"
        << "Test FAILED!" << endl;
        rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
        << "\n"
        << "Test FAILED!" << endl;
            rc = -2;
  }
  return rc;
}
```

D.1.2 Model Evaluator for ss_con1d

C++ source file

```
// ************************************************************************************************
//
// CON1D_ModelEval0.cpp
// This is a ss_con1d LIME model evaluator used for coupling with ss_neutron0
//
// ************************************************************************************************
#include <iostream>
#include "CON1D_ModelEval0.hpp"

#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

// Data from fortran modules and fortran routines are referenced using an extern C
// block as shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

 extern "C" {

  // fortran subroutines in ss_con1d.f90

   #define setup_con1d          LIME_MANGLING_GLOBAL(setup_con1d, SETUP_CON1D)
   #define finish_con1d         LIME_MANGLING_GLOBAL(finish_con1d, FINISH_CON1D)
   #define take_time_step_con1d LIME_MANGLING_GLOBAL(take_time_step_con1d, TAKE_TIME_STEP_CON1D)
   #define update_con1d         LIME_MANGLING_GLOBAL(update_con1d, UPDATE_CON1D)
   #define con1d_residual       LIME_MANGLING_GLOBAL(residual_con1d, RESIDUAL_CON1D)

   void setup_con1d();
   void finish_con1d();
   void take_time_step_con1d();
   void update_con1d();
   void con1d_residual();

   // data we will need access to from the fortran90 module "con1d_mod" in ss_con1d.f90

   #define con1d_qs    LIME_MANGLING_MODULE(con1d_mod, qs, CON1D_MOD,QS)
   #define con1d_t     LIME_MANGLING_MODULE(con1d_mod, t,  CON1D_MOD, T)
   #define con1d_tr    LIME_MANGLING_MODULE(con1d_mod, tr, CON1D_MOD, TR)
   #define con1d_dt    LIME_MANGLING_MODULE(con1d_mod, dt, CON1D_MOD, DT)
   #define con1d_tmax  LIME_MANGLING_MODULE(con1d_mod, tmax, CON1D_MOD, TMAX)
   #define con1d_time  LIME_MANGLING_MODULE(con1d_mod, time, CON1D_MOD, TIME)
   #define con1d_r_max LIME_MANGLING_MODULE(con1d_mod, r_max, CON1D_MOD, R_MAX)
   #define con1d_tol   LIME_MANGLING_MODULE(con1d_mod, tol, CON1D_MOD, TOL)

   extern float con1d_qs[3];
   extern float con1d_t[3];
   extern float con1d_tr;
```

```
   extern float con1d_dt;
   extern float con1d_tmax;
   extern float con1d_time;
   extern float con1d_r_max;
   extern float con1d_tol;
 }

//------------CON1D_ModelEval0 C++ constructor -----------------------------------------

CON1D_ModelEval0::CON1D_ModelEval0(LIME::Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name), qs(con1d_qs), t(con1d_t), trbc(con1d_tr)
{
  setup_con1d();
}

//------------CON1D_ModelEval0 C++ destructor -----------------------------------------

CON1D_ModelEval0::~CON1D_ModelEval0()
{
   finish_con1d();
}

//------------CON1D_ModelEval0 solve_standalone -----------------------------------------

void CON1D_ModelEval0::solve_standalone()
{
    // std::cout << "CON1D_ModelEval0::solve_standalone called" << std::endl;
    take_time_step_con1d();
}

//------------ CON1D_ModelEval0 Implimentation of get_time_step -------------------------

double CON1D_ModelEval0::get_time_step() const
{
   return con1d_dt;
}

//------------ CON1D_ModelEval0 Implimentation of get_max_time --------------------------

double CON1D_ModelEval0::get_max_time() const
{
   return con1d_tmax;
}

//------------ CON1D_ModelEval0 Implimentation of get_current_time ----------------------

double CON1D_ModelEval0::get_current_time() const
{
   return con1d_time;
}

//------------ CON1D_ModelEval0 Implimentation of update_time ---------------------------

void CON1D_ModelEval0::update_time()
{
   update_con1d();
}

//------------ CON1D_ModelEval0 Implimentation of is_converged --------------------------

bool CON1D_ModelEval0::is_converged()
{
    con1d_residual();
    // std::cout << "DEBUG CON1D_ModelEval0 R_max = " << con1d_r_max << std::endl;
    return( con1d_r_max < con1d_tol );
}
```

C++ header file

```
// ************************************************************************************************
//
// CON1D_ModelEval0.hpp
// This is the ss_con1d LIME model evaluator used for a simple coupling with ss_neutron0
// or ss_thinwall
//
// ************************************************************************************************
#ifndef LIME_EXAMPLE_CON1D_MODELEVAL0_HPP
#define LIME_EXAMPLE_CON1D_MODELEVAL0_HPP

#include "LIME_Model_Evaluator.hpp"

class CON1D_ModelEval0 : public LIME::Model_Evaluator
{

  public:

    CON1D_ModelEval0(LIME::Problem_Manager & pm, const string & name);   // C++ constructor

    virtual ~CON1D_ModelEval0();                                         // C++ destructor

    // Tells the LIME problem manger that this code supports a stand-alone solve
    virtual bool supports_standalone_solve() const { return true; }

    // Tells the LIME problem manger that ss_con1d solves a transient problem LIME must direct
    virtual bool is_transient() const { return true; }

    // The implementation of the following functions must be defined here or in the .cpp file
    virtual void solve_standalone();
    virtual bool is_converged();
    virtual double get_time_step() const;
    virtual double get_max_time() const;
    virtual double get_current_time() const;
    virtual unsigned int get_max_steps() const { return 50; }
    virtual void update_time();

   // data required for transfer operators that will be implemented
   float* qs; // will be transferred from ss_neutron0   to ss_con1d
   float* t;  // will be transferred   to ss_neutron0 from ss_con1d
   float& trbc; // thinwall will write this data
};
#endif
```

D.1.3 Model Evaluator for ss_neutron

C++ source file

```
// ************************************************************************************************
//
// NEUTRON_ModelEval0.cpp
// This is a ss_neutron0 LIME model evaluator used for coupling with ss_con1d
// It includes data fransfer operators needed for this coupling
//
// ************************************************************************************************
#include <iostream>

#include "NEUTRON_ModelEval0.hpp"
#include "CON1D_ModelEval0.hpp"

#include "LIME_Problem_Manager.hpp"
```

```
#include "LIME_fortran_mangling.h"

using LIME::Data_Transfer_Operator;
using LIME::Model_Evaluator;
using LIME::Problem_Manager;

// Data from fortran modules and fortran routines are referenced using an extern C
// block as shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

extern "C" {

   // fortran90 module data that we need access to

  #define neutron0_q     LIME_MANGLING_MODULE(neutron0_mod,q, NEUTRON0_MOD,Q)
  #define neutron0_t     LIME_MANGLING_MODULE(neutron0_mod,t, NEUTRON0_MOD,T)
  #define neutron0_r_max LIME_MANGLING_MODULE(neutron0_mod,r_max, NEUTRON0_MOD,R_MAX)

  extern float neutron0_q[4];
  extern float neutron0_t[4];
  extern float neutron0_r_max;

  // fortran90 subroutines that we need to call

  #define setup_neutron0  LIME_MANGLING_GLOBAL(setup_neutron0, SETUP_NEUTRON0)
  #define solve_neutron0  LIME_MANGLING_GLOBAL(solve_neutron0, SOLVE_NEUTRON0)
  #define finish_neutron0 LIME_MANGLING_GLOBAL(finish_neutron0, FINISH_NEUTRON0)

  void setup_neutron0();
  void solve_neutron0();
  void finish_neutron0();
}

//------------NEUTRON_ModelEval0 C++ constructor -----------------------------------------

NEUTRON_ModelEval0::NEUTRON_ModelEval0(Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name), q(neutron0_q), t(neutron0_t)
{
  setup_neutron0();
}

//-------------NEUTRON_ModelEval0 solve_standalone ----------------------------------------

void NEUTRON_ModelEval0::solve_standalone()
{
  //std::cout << "NEUTRON_ModelEval0::solve_standalone() called" << std::endl;
    solve_neutron0();
}

//------------NEUTRON_ModelEval0 C++ destructor -----------------------------------------

NEUTRON_ModelEval0::~NEUTRON_ModelEval0()
{
   finish_neutron0();
}

//------------ NEUTRON_ModelEval0 Implimentation of perform_elimination ------------------------

bool NEUTRON_ModelEval0::perform_elimination()
{
   solve_neutron0();
   return true;
}


//-------------------------------------------------------------------------------------------
// The required data transfer operators are included next
//-------------------------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------------------------
conduction_2_neutronics::conduction_2_neutronics(Teuchos::RCP<Model_Evaluator> from,
                                                 Teuchos::RCP<Model_Evaluator> to)
      : Data_Transfer_Operator(from, to)
{
}

//---------------------------------------------------------------------------------------------
bool conduction_2_neutronics::perform_data_transfer() const
{
   const Teuchos::RCP<CON1D_ModelEval0> & con1d_me = Teuchos::rcp_dynamic_cast<CON1D_ModelEval0>(source());
   const Teuchos::RCP<NEUTRON_ModelEval0> & neutron_me = Teuchos::rcp_dynamic_cast<NEUTRON_ModelEval0>(target());

  std::copy(con1d_me->t, con1d_me->t+3, neutron_me->t);
  //std::cout << "conduction_2_neutronics transfer" << std::endl;
  //std::cout << neutron_me->t[1] << std::endl;

  return true;
}

//---------------------------------------------------------------------------------------------
neutronics_2_conduction::neutronics_2_conduction(Teuchos::RCP<Model_Evaluator> from,
                                                 Teuchos::RCP<Model_Evaluator> to)
      : Data_Transfer_Operator(from, to)
{ }

//---------------------------------------------------------------------------------------------
bool neutronics_2_conduction::perform_data_transfer() const
{
   const Teuchos::RCP<NEUTRON_ModelEval0> & neutron_me = Teuchos::rcp_dynamic_cast<NEUTRON_ModelEval0>(source());
   const Teuchos::RCP<CON1D_ModelEval0> & con1d_me = Teuchos::rcp_dynamic_cast<CON1D_ModelEval0>(target());

   std::copy(neutron_me->q, neutron_me->q+3, con1d_me->qs);
   //std::cout << "neutronics_2_conduction transfer" << std::endl;
   //std::cout << con1d_me->qs[1] << std::endl;
   //std::cout << neutron_me->q[0] << ", " << neutron_me->q[1] << ", " << neutron_me->q[2] << ", " << con1d_me->qs[1] << std::er

  return true;
}
```

C++ header file

```
// ********************************************************************************************
//
// NEUTRON_ModelEval0.hpp
// This is the ss_neutron LIME model evaluator used for a simple coupling with ss_con1d
//
// ********************************************************************************************
#ifndef LIME_EXAMPLE_NEUTRON_MODELEVAL1_HPP
#define LIME_EXAMPLE_NEUTRON_MODELEVAL1_HPP

#include "LIME_Model_Evaluator.hpp"
#include "LIME_Elimination_Module.hpp"
#include "LIME_Data_Transfer_Operator.hpp"

class NEUTRON_ModelEval0 : public LIME::Model_Evaluator, public Elimination_Module
{

  public:

    NEUTRON_ModelEval0(LIME::Problem_Manager & pm, const string & name);    // C++ constructor
```

```
    virtual ~NEUTRON_ModelEval0();                                          // C++ destructor


    // The implementation of the following functions are defined here in the .cpp file
    virtual bool perform_elimination();
    virtual void solve_standalone();

   // data required for transfer operators that will be implemented
    float *q, *t;

};


// -- ss_con1d to ss_neutron transfer operator -------
//
class conduction_2_neutronics : public LIME::Data_Transfer_Operator {
public:
    conduction_2_neutronics(Teuchos::RCP<LIME::Model_Evaluator> from, Teuchos::RCP<LIME::Model_Evaluator> to);

    // The implimentation of this function is defined in the c++ source file
    virtual bool perform_data_transfer() const;
};


// -- ss_neutron to ss_con1d transfer operator -------
//
class neutronics_2_conduction : public LIME::Data_Transfer_Operator {
public:
    neutronics_2_conduction(Teuchos::RCP<LIME::Model_Evaluator> from, Teuchos::RCP<LIME::Model_Evaluator> to);

    // The implimentation of this function is defined in the c++ source file
    virtual bool perform_data_transfer() const;
};
#endif
```

D.2 Example 2. Fixed-Point Coupling of ss_con1d and ss_thinwall using local convergence checks

This appendix provides a listing of the MPDriver source code, as well as the C++ source file and C++ header file that comprise the Model Evaluator for ss_thinwall for Example 2. Note that the Model Evaluator source code for the ss_con1d Model evaluator used here is unchanged from Example 1, and thus is not listed again.

D.2.1 Multiphysics Driver for Example 2

```
// ************************************************************************************************
//
// lime_mpd1.cpp
// This is a LIME driver for fixed pt coupling of super-simple codes ss_con1d and ss_thinwall
// where convergence is determined by each code meeting its own requirments
// The following Model Evaluators are used:
//            CON1D_ModelEval0,         and THINWALL_ModelEval0
//
// ************************************************************************************************
//
// "#include" directives for the various header files that are needed
```

```
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d and thinwall physics
#include "CON1D_ModelEval0.hpp"
#include "THINWALL_ModelEval0.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
          LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create pointers to an instance of a "CON1D_ModelEval0" object called "con1d"
    //    and an instance of a "THINWALL_ModelEval0" object called "thinwall"
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

          Teuchos::RCP<CON1D_ModelEval0> con1d =
                   Teuchos::rcp(new CON1D_ModelEval0(pm,"CON1DModelEval0"));

          Teuchos::RCP<THINWALL_ModelEval0> thinwall =
                   Teuchos::rcp(new THINWALL_ModelEval0(pm,"THINWALLModelEval0"));

    // 4  Register or "add" our physics problems "con1d" and "thinwall" with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "con1d_id" and "thinwall_id" from the Problem Manager.

          pm.add_problem(con1d);
          pm.add_problem(thinwall);

    // 5  Create and setup the two LIME data transfer operators that we will need
```

```
        LIME::Data_Transfer_Operator* p_t2c = new thinwall_2_conduction(thinwall, con1d);
        LIME::Data_Transfer_Operator* p_c2t = new conduction_2_thinwall(con1d, thinwall);

        Teuchos::RCP<LIME::Data_Transfer_Operator> t2c_op = Teuchos::rcp(p_t2c);
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2t_op = Teuchos::rcp(p_c2t);


    // 6  Register or "add" our two transfer operators "t2c_op" and "c2t_op" with
    //    the Problem manager

        pm.add_preelimination_transfer(c2t_op);
        pm.add_postelimination_transfer(t2c_op);

        pm.register_complete(); // Trigger setup of groups, solvers, etc.


    // 7  We are now ready to let the problem manager drive the coupled problem

         pm.integrate();
  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
          rc = -2;
  }

  return rc;
}
```

D.2.2 Model Evaluator for ss_thinwall

C++ source file

```
// ************************************************************************************
//
// THINWALL_ModelEval0.cpp
// This is the ss_thinwall LIME model evaluator used for a simple coupling with ss_con1d
//
// ************************************************************************************
#include "THINWALL_ModelEval0.hpp"
#include "CON1D_ModelEval0.hpp"

#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Data_Transfer_Operator;
using LIME::Model_Evaluator;
using LIME::Problem_Manager;

// Data from fortran modules and fortran routines are referenced using an extern C
// block as shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

 extern "C" {
```

```
   // fortran module data

  #define thinwall_t    LIME_MANGLING_MODULE(thinwall_mod, t   , THINWALL_MOD, T   )
  #define thinwall_trad LIME_MANGLING_MODULE(thinwall_mod, trad, THINWALL_MOD, TRAD)
  #define thinwall_dt   LIME_MANGLING_MODULE(thinwall_mod, dt  , THINWALL_MOD, DT  )
  #define thinwall_time LIME_MANGLING_MODULE(thinwall_mod, time, THINWALL_MOD, TIME)
  #define thinwall_r    LIME_MANGLING_MODULE(thinwall_mod, r   , THINWALL_MOD, R   )
  #define thinwall_tol  LIME_MANGLING_MODULE(thinwall_mod, tol , THINWALL_MOD, TOL )

  extern float thinwall_t[1];   //     thinwall temperature
  extern float thinwall_trad;   //     thinwall radiation BC temperature
  extern float thinwall_dt;     //     solution time step
  extern float thinwall_time;   //     current simulation time
  extern float thinwall_r[1];   //     residual value
  extern float thinwall_tol ;   //     convergence tolerance

  #define VAR_T     thinwall_t
  #define VAR_TRAD  thinwall_trad
  #define VAR_DT    thinwall_dt
  #define VAR_TMAX  100000.0               //  hardwired to large value
  #define VAR_TIME  thinwall_time
  #define VAR_R_MAX thinwall_r
  #define VAR_TOL   thinwall_tol

  // fortran subroutines

  #define setup_thinwall     LIME_MANGLING_GLOBAL(setup_thinwall, SETUP_THINWALL)
  #define finish_thinwall    LIME_MANGLING_GLOBAL(finish_thinwall, FINISH_THINWALL)
  #define time_step_thinwall LIME_MANGLING_GLOBAL(time_step_thinwall, TIME_STEP_THINWALL)
  #define update_thinwall    LIME_MANGLING_GLOBAL(update_thinwall, UPDATE_THINWALL)
  #define residual_thinwall  LIME_MANGLING_GLOBAL(residual_thinwall, RESIDUAL_THINWALL)

   void setup_thinwall();
   void finish_thinwall();
   void time_step_thinwall();
   void update_thinwall();
   void residual_thinwall();
 }

// Set the "correct" gold results
// const float THINWALL_ModelEval0::gold_w_temp = 346.6981;

//------------THINWALL ModelEval0 C++ constructor  -----------------------------------------------

THINWALL_ModelEval0::THINWALL_ModelEval0(Problem_Manager & pm, const string & name) :
  Model_Evaluator(pm, name), t(VAR_T), trad(VAR_TRAD)
{
  setup_thinwall();
}

//------------THINWALL ModelEval0 C++ destructor -----------------------------------------------

THINWALL_ModelEval0::~THINWALL_ModelEval0()
{
   finish_thinwall();
}


//------------THINWALL_ModelEval0 Implimentation of solve_standalone --------------------------

void THINWALL_ModelEval0::solve_standalone()
{
   //std::cout << "THINWALL_ModelEval0::solve_standalone called" << std::endl;
   time_step_thinwall();
}

//------------THINWALL_ModelEval0 Implimentation of is_converged ------------------------------
```

```
bool THINWALL_ModelEval0::is_converged()
{
  residual_thinwall();
  //std::cout << "DEBUG THINWALL_ModelEval0, VAR_R_MAX=" << VAR_R_MAX[0] << std::endl;
    return( VAR_R_MAX[0] < VAR_TOL );
}


//------------THINWALL_ModelEval0 Implimentation of get_time_step ---------------------------

double THINWALL_ModelEval0::get_time_step() const
{
   return VAR_DT;
}


//------------THINWALL_ModelEval0 Implimentation of get_max_time ----------------------------

double THINWALL_ModelEval0::get_max_time() const
{
   return VAR_TMAX;
}

//------------THINWALL_ModelEval0 Implimentation of get_current_time -------------------------

double THINWALL_ModelEval0::get_current_time() const
{
   return VAR_TIME;
}

//------------THINWALL_ModelEval0 Implimentation of update_time ------------------------------

void THINWALL_ModelEval0::update_time()
{
   update_thinwall();
}


//-----------------------------------------------------------------------------------------
// The required data transfer operators are included next
//-----------------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------------
conduction_2_thinwall::conduction_2_thinwall(Teuchos::RCP<Model_Evaluator> from,
                                             Teuchos::RCP<Model_Evaluator> to)
      : Data_Transfer_Operator(from, to)
{
}

//-----------------------------------------------------------------------------------------
bool conduction_2_thinwall::perform_data_transfer() const
{
   const Teuchos::RCP<CON1D_ModelEval0> & con1d_me = Teuchos::rcp_dynamic_cast<CON1D_ModelEval0>(source());
   const Teuchos::RCP<THINWALL_ModelEval0> & thinwall_me = Teuchos::rcp_dynamic_cast<THINWALL_ModelEval0>(target());

  thinwall_me->trad = con1d_me->t[2];
  //std::cout << "conduction_2_thinwall transfer" << std::endl;
  //std::cout << thinwall_me->trad << std::endl;

  return true;
}

//-----------------------------------------------------------------------------------------
thinwall_2_conduction::thinwall_2_conduction(Teuchos::RCP<Model_Evaluator> from,
                                             Teuchos::RCP<Model_Evaluator> to)
      : Data_Transfer_Operator(from, to)
```

```
{ }

//-------------------------------------------------------------------------------------------
bool thinwall_2_conduction::perform_data_transfer() const
{
   const Teuchos::RCP<THINWALL_ModelEval0> & thinwall_me = Teuchos::rcp_dynamic_cast<THINWALL_ModelEval0>(source());
   const Teuchos::RCP<CON1D_ModelEval0> & con1d_me = Teuchos::rcp_dynamic_cast<CON1D_ModelEval0>(target());

   con1d_me->trbc = thinwall_me->t[0];
   //std::cout << "thinwall_2_conduction transfer" << std::endl;
   //std::cout << con1d_me->trbc << std::endl;

  return true;
}
```

C++ header file

```
// ************************************************************************************************
//
// THINWALL_ModelEval0.hpp
// This is the ss_thinwall LIME model evaluator used for a simple coupling with ss_con1d
//
// ************************************************************************************************
#ifndef LIME_EXAMPLE_THINWALL_MODELEVAL0_HPP
#define LIME_EXAMPLE_THINWALL_MODELEVAL0_HPP

#include "LIME_Model_Evaluator.hpp"
#include "LIME_Data_Transfer_Operator.hpp"

class THINWALL_ModelEval0 : public LIME::Model_Evaluator
{

  public:

    THINWALL_ModelEval0(LIME::Problem_Manager & pm, const string & name);   // C++ constructor

    virtual ~THINWALL_ModelEval0();                                         // C++ destructor

    // Tells the LIME problem manger that this code supports a stand-alone solve
    virtual bool supports_standalone_solve() const { return true; }

    // Tells the LIME problem manger that ss_con1d solves a transient problem LIME must direct
    virtual bool is_transient() const { return true; }

    // The implementation of the following functions must be defined here or in the .cpp file
    virtual void solve_standalone();
    virtual bool is_converged();
    virtual double get_time_step() const;
    virtual double get_max_time() const;
    virtual double get_current_time() const;
    virtual unsigned int get_max_steps() const { return 50; }
    virtual void update_time();

   // data required for transfer operators that will be implemented
   float* t;       // will transfer this array   to con1d from thinwall
   float& trad;    // will transfer this value from con1d   to thinwall

};


// -- ss_con1d to ss_thinwall transfer operator -------
//
class conduction_2_thinwall : public LIME::Data_Transfer_Operator {
public:
    conduction_2_thinwall(Teuchos::RCP<LIME::Model_Evaluator> from, Teuchos::RCP<LIME::Model_Evaluator> to);
```

```
    // The implimentation of this function is defined in the c++ source file
    virtual bool perform_data_transfer() const;
};


// -- ss_thinwall to ss_con1d transfer operator -------
//
class thinwall_2_conduction : public LIME::Data_Transfer_Operator {
public:
    thinwall_2_conduction(Teuchos::RCP<LIME::Model_Evaluator> from, Teuchos::RCP<LIME::Model_Evaluator> to);

    // The implimentation of this function is defined in the c++ source file
    virtual bool perform_data_transfer() const;
};
#endif
```

D.3 Example 3. Fixed-Point Coupling of ss_con1d, ss_neutron and ss_thinwall

This appendix provides a listing of the MPDriver source code for Example 3. The C++ source files and C++ header files for each of the three physics codes that are coupled here are identical to those used previously for Example Problems 1 and 2, and thus are not repeated here.

D.3.1 Multiphysics Driver for Example 3

```
// ************************************************************************************************
//
// lime_mpd_all.cpp
// This is a LIME driver for coupling the "super-simple" codes ss_con1d, ss_neutron0, and
// ss_ thinwall  using the following Model Evaluators:
//             CON1D_ModelEval0, NEUTRON_ModelEval0, and THINWALL_ModelEval0
//
// ************************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d physics header file
#include "CON1D_ModelEval0.hpp"

// neutron physics
#include "NEUTRON_ModelEval0.hpp"

// thinwall physics header file
#include "THINWALL_ModelEval0.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
```

```
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
          LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create pointers to
    //    an instance of a "CON1D_ModelEval0"    object called "con1d", and
    //    an instance of a "NEUTRON_ModelEval0"  object called "neutron", and
    //    an instance of a "THINWALL_ModelEval0" object called "thinwall"
    //    (on the heap)
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

          Teuchos::RCP<CON1D_ModelEval0> con1d =
                   Teuchos::rcp(new CON1D_ModelEval0(pm,"CON1DModelEval0"));

          Teuchos::RCP<NEUTRON_ModelEval0> neutron =
                   Teuchos::rcp(new NEUTRON_ModelEval0(pm,"NEUTRONModelEval0"));

          Teuchos::RCP<THINWALL_ModelEval0> thinwall =
                   Teuchos::rcp(new THINWALL_ModelEval0(pm,"THINWALLModelEval0"));

    // 4  Register or "add" our physics problems "con1d", "neutron", and "thinwall" with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "con1d_id", "neutron_id" and "thinwall_id" from the Problem Manager.

          pm.add_problem(con1d);
          pm.add_problem(neutron);
          pm.add_problem(thinwall);

    // 5  Create and setup the four LIME data transfer operators that we will need

          LIME::Data_Transfer_Operator* p_n2c = new neutronics_2_conduction(neutron, con1d);
          LIME::Data_Transfer_Operator* p_c2n = new conduction_2_neutronics(con1d, neutron);
          LIME::Data_Transfer_Operator* p_t2c = new thinwall_2_conduction(thinwall, con1d);
          LIME::Data_Transfer_Operator* p_c2t = new conduction_2_thinwall(con1d, thinwall);

          Teuchos::RCP<LIME::Data_Transfer_Operator> n2c_op = Teuchos::rcp(p_n2c);
```

```
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2n_op = Teuchos::rcp(p_c2n);
        Teuchos::RCP<LIME::Data_Transfer_Operator> t2c_op = Teuchos::rcp(p_t2c);
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2t_op = Teuchos::rcp(p_c2t);


   // 6  Register or "add" our four transfer operators with
   //    the Problem manager

        pm.add_preelimination_transfer(c2n_op);
        pm.add_preelimination_transfer(c2t_op);
        pm.add_postelimination_transfer(n2c_op);
        pm.add_postelimination_transfer(t2c_op);

        pm.register_complete();  // Trigger setup of groups, solvers, etc.


   // 7  We are now ready to let the problem manager drive the coupled problem

         pm.integrate();
  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
          rc = -2;
  }

  return rc;
}
```

D.4 Example 4. Fixed-Point Coupling of ss_con1d and ss_thinwall using a global residual based convergence check

This appendix provides a listing of the MPDriver source code, as well as the C++ source files and C++ header files that extend the Model Evaluators for ss_con1d and ss_thinwall as needed for this example.

D.4.1 Multiphysics Driver for Example 4

```
// ************************************************************************************************
//
// lime_mpd1B.cpp
// This is a LIME driver for fixed pt coupling of super-simple codes ss_con1d and ss_thinwall
// where convergence is determined using a global residual based convergence check.
// The following Model Evaluators are used (note the last 2 are used indirectly):
//            CON1D_ModelEval0_w_Resid, and THINWALL_ModelEval0_w_Resid
//            CON1D_ModelEval0,         and THINWALL_ModelEval0
```

```
// ***********************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d and thinwall physics header files
#include "CON1D_ModelEval0_w_Resid.hpp"
#include "THINWALL_ModelEval0_w_Resid.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// -----------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors

  try {    // the "try" construct is used to allow for catching "exceptions" that might
           // unexpectedly occur.  They can occur at any point in the program's
           // call stack so our exception handling policy is to catch all thrown excptions
           // here in main, output a diagnostic message, and terminate the program cleanly.


    // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
    //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
    //    must be constructed with either an MPI communicator or, if we are not using MPI, a
    //    class that has the same interface as the MPI communicator but does nothing.

          Epetra_Comm* comm = new Epetra_SerialComm;

    // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
    //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

          bool verbose = false;
          LIME::Problem_Manager pm(*comm, verbose);


    // 3  Create pointers to an instance of a "CON1D_ModelEval0_w_Resid" object called "con1d"
    //    and an instance of a "THINWALL_ModelEval0_w_Resid" object called "thinwall"
    //    Note: There are some Trilinos-specific implementation details here. For example,
    //          Teuchos is a Trilinos library and RCP is a reference counted pointer

          Teuchos::RCP<CON1D_ModelEval0_w_Resid> con1d =
                   Teuchos::rcp(new CON1D_ModelEval0_w_Resid(pm,"CON1DModelEval0_w_Resid"));

          Teuchos::RCP<THINWALL_ModelEval0_w_Resid> thinwall =
                   Teuchos::rcp(new THINWALL_ModelEval0_w_Resid(pm,"THINWALLModelEval0_w_Resid"));

    // 4  Register or "add" our physics problems "con1d" and "thinwall" with
    //    the Problem manager. When doing this, we get back values for the
    //    identifier "con1d_id" and "thinwall_id" from the Problem Manager.

          pm.add_problem(con1d);
```

```
          pm.add_problem(thinwall);

    // 5  Create and setup the two LIME data transfer operators that we will need

          LIME::Data_Transfer_Operator* p_t2c = new thinwall_2_conduction(thinwall, con1d);
          LIME::Data_Transfer_Operator* p_c2t = new conduction_2_thinwall(con1d, thinwall);

          Teuchos::RCP<LIME::Data_Transfer_Operator> t2c_op = Teuchos::rcp(p_t2c);
          Teuchos::RCP<LIME::Data_Transfer_Operator> c2t_op = Teuchos::rcp(p_c2t);


    // 6  Register or "add" our two transfer operators "t2c_op" and "c2t_op" with
    //    the Problem manager

          pm.add_preelimination_transfer(c2t_op);
          pm.add_postelimination_transfer(t2c_op);

          pm.register_complete(); // Trigger setup of groups, solvers, etc.


    // 7  We are now ready to let the problem manager drive the coupled problem

           pm.integrate();
  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
             rc = -2;
  }

  return rc;
}
```

D.4.2 Extended Model Evaluator for ss_con1d used in Example 4

C++ source file

```
// ************************************************************************************************
//
// CON1D_ModelEval0_w_Resid.cpp
// This is an additional member function that inherits from the main LIME model evaluator
// for a stand-alone wrap of the super-simple con1d code, and adds the functionality needed
// for passing a residual up to LIME for evaluation
//
// ************************************************************************************************
#include <iostream>
#include "CON1D_ModelEval0_w_Resid.hpp"

#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Problem_Manager;

// Data from fortran modules and fortran routines are referenced using an extern C
// block as shown below. The "name mangling" conventions are taken care of automatically
```

```
// in LIME by #including the file LIME_fortran_mangling.h

extern "C" {

  // fortran module data

  #define con1d_qs    LIME_MANGLING_MODULE(con1d_mod, qs, CON1D_MOD, QS)
  #define con1d_t     LIME_MANGLING_MODULE(con1d_mod, t, CON1D_MOD, T)
  #define con1d_r     LIME_MANGLING_MODULE(con1d_mod, r, CON1D_MOD, R)
  #define con1d_dt    LIME_MANGLING_MODULE(con1d_mod, dt, CON1D_MOD, dt)
  #define con1d_tmax  LIME_MANGLING_MODULE(con1d_mod, tmax, CON1D_MOD, TMAX)
  #define con1d_time  LIME_MANGLING_MODULE(con1d_mod, time, CON1D_MOD, TIME)
  #define con1d_r_max LIME_MANGLING_MODULE(con1d_mod, r_max, CON1D_MOD, R_MAX)
  #define con1d_tol   LIME_MANGLING_MODULE(con1d_mod, tol, CON1D_MOD, TOL)
  #define con1d_nvars LIME_MANGLING_MODULE(con1d_mod, n_vars, CON1D_MOD, N_VARS)

  extern float con1d_qs[3];
  extern float con1d_t[3];
  extern float con1d_r[3];
  extern float con1d_dt;
  extern float con1d_tmax;
  extern float con1d_time;
  extern float con1d_r_max;
  extern float con1d_tol;
  extern int con1d_nvars;

  // fortran subroutines

  #define setup_con1d          LIME_MANGLING_GLOBAL(setup_con1d, SETUP_CON1D)
  #define finish_con1d         LIME_MANGLING_GLOBAL(finish_con1d, FINISH_CON1D)
  #define take_time_step_con1d LIME_MANGLING_GLOBAL(take_time_step_con1d, TAKE_TIME_STEP_CON1D)
  #define update_con1d         LIME_MANGLING_GLOBAL(update_con1d, UPDATE_CON1D)
  #define con1d_residual       LIME_MANGLING_GLOBAL(residual_con1d, RESIDUAL_CON1D)

  void setup_con1d();
  void finish_con1d();
  void take_time_step_con1d();
  void update_con1d();
  void con1d_residual();

}

//------------CON1D_w_Resid constructor -------------------------------------------------

CON1D_ModelEval0_w_Resid::CON1D_ModelEval0_w_Resid(Problem_Manager & pm, const string & name) :
  CON1D_ModelEval0(pm, name), r(con1d_r)
{
  // Create an Epetra Map decribing the data layout for both the
  // unknowns array and residuals array.  This is trivial in serial
  // but contains distribution info in parallel.
  epetra_map_ = Teuchos::rcp(new Epetra_Map(con1d_nvars, 0, pm.Comm()));

  // Create a state vector conforming to the data layout map
  // This vector gets used to create conformal solver objects by the solvers
  // as well as to convey initial values for time stepping and nonlinear
  // iterations
  me_interface_soln_ = Teuchos::rcp(new Epetra_Vector(*epetra_map_));
}

// ------------CON1D_w_Resid get_x_init -------------------------------------------------

Teuchos::RCP<const Epetra_Vector>
CON1D_ModelEval0_w_Resid::get_x_init() const
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
```

```
}

// ------------CON1D_w_Resid get_x_state ------------------------------------------------

Teuchos::RCP<Epetra_Vector>
CON1D_ModelEval0_w_Resid::get_x_state()
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
}

// ------------CON1D_w_Resid get_x_state ------------------------------------------------

Teuchos::RCP<Epetra_Vector>
CON1D_ModelEval0_w_Resid::get_x_state() const
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
}

//------------CON1D_w_Resid initializeSolution ----------------------------------------------

void
CON1D_ModelEval0_w_Resid::initializeSolution()
{
  // Copy initial values from our wrapped application
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
}

//------------CON1D_w_Resid createInArgs ----------------------------------------------

EpetraExt::ModelEvaluator::InArgs
CON1D_ModelEval0_w_Resid::createInArgs() const
{
  EpetraExt::ModelEvaluator::InArgsSetup inArgs;

  // Register our identify
  inArgs.setModelEvalDescription(my_name_);

  // Signal that we can do calculations with incoming state x
  inArgs.setSupports(IN_ARG_x, true);

  return inArgs;
}

//------------CON1D_w_Resid createOutArgs ----------------------------------------------

EpetraExt::ModelEvaluator::OutArgs
CON1D_ModelEval0_w_Resid::createOutArgs() const
{
  EpetraExt::ModelEvaluator::OutArgsSetup outArgs;

  // Register our identify - consistent with createInArgs
  outArgs.setModelEvalDescription(my_name_);

  // Signal that we can compute a residual vector
  outArgs.setSupports(OUT_ARG_f, true);

  return outArgs;
}

//------------CON1D_w_Resid evalModel ----------------------------------------------
```

```
void
CON1D_ModelEval0_w_Resid::evalModel(const InArgs& inArgs, const OutArgs& outArgs) const
{
  // Create a "View" of the incoming solution state, x
  //  This avoids a copy of data and becomes important for real-sized problems
  const Epetra_Vector x(View, *(inArgs.get_x().get()), 0);

  //cout << "CON1D_ModelEval0_w_Resid::evalModel called. state:\n";

  // Copy values into our wrapped application's data array
  for( int i = 0; i < x.MyLength(); ++i )
  {
    //cout << "x[" << i << "] = " << x[i] << endl;
    (this->t)[i] = x[i];
  }

  if( outArgs.get_f().get() ) // A non-NULL f-vector signals a residual fill request
  {
    // Get a reference to the vector we will populate
    Epetra_Vector & f = *(outArgs.get_f().get());

    // compute our application's nonlinear residual
    residual_con1d_();

    // Copy application residual array into f-vector provided by outArgs
    for( int i = 0; i < f.MyLength(); ++i )
    {
      f[i] = (this->r)[i];
      //cout << "f[" << i << "] = " << f[i] << endl;
    }
  }

  // We don't yet support this and respond to such a request with an error
  if( outArgs.get_W().get() ) // Signals either a computeJacobian or a computePreconditioner
    throw std::runtime_error("CON1D_ModelEval0_w_Resid::evalModel : Jacobian matrix support not available.");
}
```

C++ header file

```
// ***********************************************************************************************
//
// CON1D_ModelEval0_w_Resid.hpp
// This is the ss_con1d LIME model evaluator used for a simple coupling with ss_thinwall
// when basing convergence on a global residual check
//
// ***********************************************************************************************
#ifndef LIME_EXAMPLE_CON1D_W_RESID_MODELEVAL_HPP
#define LIME_EXAMPLE_CON1D_W_RESID_MODELEVAL_HPP

#include "CON1D_ModelEval0.hpp"

class CON1D_ModelEval0_w_Resid : public CON1D_ModelEval0
{
  public:

    CON1D_ModelEval0_w_Resid(LIME::Problem_Manager & pm, const string & name); // C++ constructor

    virtual ~CON1D_ModelEval0_w_Resid() {};                                    // C++ destructor


    // Methods in addition to those in CON1D_ModelEval0 that support residual fill callbacks
    virtual EpetraExt::ModelEvaluator::InArgs createInArgs() const;
    virtual EpetraExt::ModelEvaluator::OutArgs createOutArgs() const;
    virtual void evalModel(const InArgs&, const OutArgs&) const;
```

```
    virtual Teuchos::RCP<const Epetra_Vector> get_x_init() const;
    virtual Teuchos::RCP<Epetra_Vector> get_x_state();
    virtual Teuchos::RCP<Epetra_Vector> get_x_state() const;

    void initializeSolution();

    // This should be removed in favor of querying via createInArgs
    virtual bool supports_residual() const { return true; }


  protected:

    // A data layour map needed to create/configure solver objects
    Teuchos::RCP<Epetra_Map> epetra_map_;

    // The ME state vector conformal to the data layout map and that is
    // also used to create/configure solver objects
    mutable Teuchos::RCP<Epetra_Vector> me_interface_soln_;

    // needed to get residual array values
    float* r;
};
#endif
```

D.4.3 Extended Model Evaluator for ss_thinwall used in Example 4

C++ source file

```
// ************************************************************************************************
//
// THINWALL_ModelEval0_w_Resid.cpp
// This is an additional member function that inherits from the main LIME model evaluator
// for a stand-alone wrap of the super-simple thinwall code, and adds the functionality needed
// for passing a residual up to LIME for evaluation
//
// ************************************************************************************************
#include <iostream>
#include "THINWALL_ModelEval0_w_Resid.hpp"

#include "LIME_Problem_Manager.hpp"
#include "LIME_fortran_mangling.h"

using LIME::Problem_Manager;


// Data from fortran modules and fortran routines are referenced using an extern C
// block as shown below. The "name mangling" conventions are taken care of automatically
// in LIME by #including the file LIME_fortran_mangling.h

extern "C" {

  // fortran module data

   #define thinwall_r      LIME_MANGLING_MODULE(thinwall_mod, r, THINWALL_MOD, R)
   #define thinwall_nvars  LIME_MANGLING_MODULE(thinwall_mod, n_vars, THINWALL_MOD, N_VARS)

   extern float thinwall_r[1];
   extern int   thinwall_nvars;

  // fortran subroutines

   #define residual_thinwall LIME_MANGLING_GLOBAL(residual_thinwall, RESIDUAL_THINWALL)
```

```
   void residual_thinwall();

 }

//------------THINWALL_w_Resid constructor -----------------------------------------

THINWALL_ModelEval0_w_Resid::THINWALL_ModelEval0_w_Resid(Problem_Manager & pm, const string & name) :
  THINWALL_ModelEval0(pm, name), r(thinwall_r)
{
  // Create an Epetra Map decribing the data layout for both the
  // unknowns array and residuals array.  This is trivial in serial
  // but contains distribution info in parallel.
  epetra_map_ = Teuchos::rcp(new Epetra_Map(thinwall_nvars, 0, pm.Comm()));

  // Create a state vector conforming to the data layout map
  // This vector gets used to create conformal solver objects by the solvers
  // as well as to convey initial values for time stepping and nonlinear
  // iterations
  me_interface_soln_ = Teuchos::rcp(new Epetra_Vector(*epetra_map_));
}

// ------------THINWALL_w_Resid get_x_init -------------------------------------------------

Teuchos::RCP<const Epetra_Vector>
THINWALL_ModelEval0_w_Resid::get_x_init() const
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
}

// ------------THINWALL_w_Resid get_x_state -------------------------------------------------

Teuchos::RCP<Epetra_Vector>
THINWALL_ModelEval0_w_Resid::get_x_state()
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
}

// ------------THINWALL_w_Resid get_x_state -------------------------------------------------

Teuchos::RCP<Epetra_Vector>
THINWALL_ModelEval0_w_Resid::get_x_state() const
{
   // Copy current values from our wrapped application into LIME array
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
   return me_interface_soln_;
}

//------------THINWALL_w_Resid initializeSolution -----------------------------------------

void
THINWALL_ModelEval0_w_Resid::initializeSolution()
{
  // Copy initial values from our wrapped application
  for( int i = 0; i < me_interface_soln_->MyLength(); ++i )
    (*me_interface_soln_)[i] = (this->t)[i];
}

//------------THINWALL_w_Resid createInArgs -----------------------------------------

EpetraExt::ModelEvaluator::InArgs
THINWALL_ModelEval0_w_Resid::createInArgs() const
```

```
{
  EpetraExt::ModelEvaluator::InArgsSetup inArgs;

  // Register our identify
  inArgs.setModelEvalDescription(my_name_);

  // Signal that we can do calculations with incoming state x
  inArgs.setSupports(IN_ARG_x, true);

  return inArgs;
}

//------------THINWALL_w_Resid createOutArgs ----------------------------------------------

EpetraExt::ModelEvaluator::OutArgs
THINWALL_ModelEval0_w_Resid::createOutArgs() const
{
  EpetraExt::ModelEvaluator::OutArgsSetup outArgs;

  // Register our identify - consistent with createInArgs
  outArgs.setModelEvalDescription(my_name_);

  // Signal that we can compute a residual vector
  outArgs.setSupports(OUT_ARG_f, true);

  return outArgs;
}

//------------THINWALL_w_Resid evalModel ---------------------------------------------

void
THINWALL_ModelEval0_w_Resid::evalModel(const InArgs& inArgs, const OutArgs& outArgs) const
{
  // Create a "View" of the incoming solution state, x
  //  This avoids a copy of data and becomes important for real-sized problems
  const Epetra_Vector x(View, *(inArgs.get_x().get()), 0);

  //cout << "THINWALL_ModelEval0_w_Resid::evalModel called. state:\n";

  // Copy values into our wrapped application's data array
  for( int i = 0; i < x.MyLength(); ++i )
  {
    //cout << "x[" << i << "] = " << x[i] << endl;
    (this->t)[i] = x[i];
  }

  if( outArgs.get_f().get() ) // A non-NULL f-vector signals a residual fill request
  {
    // Get a reference to the vector we will populate
    Epetra_Vector & f = *(outArgs.get_f().get());

    // compute our application's nonlinear residual
    residual_thinwall();

    // Copy application residual array into f-vector provided by outArgs
    for( int i = 0; i < f.MyLength(); ++i )
    {
      f[i] = (this->r)[i];
      //cout << "f[" << i << "] = " << f[i] << endl;
    }
  }

  // We don't yet support this and respond to such a request with an error
  if( outArgs.get_W().get() ) // Signals either a computeJacobian or a computePreconditioner
    throw std::runtime_error("THINWALL_ModelEval0_w_Resid::evalModel : Jacobian matrix support not available.");
}
```

C++ header file

```
// ************************************************************************************************
//
// THINWALL_ModelEval0_w_Resid.hpp
// This is the ss_thinwall LIME model evaluator used for a simple coupling with ss_con1d
// when basing convergence on a global residual check
//
// ************************************************************************************************
#ifndef LIME_EXAMPLE_THINWALL_W_RESID_MODELEVAL_HPP
#define LIME_EXAMPLE_THINWALL_W_RESID_MODELEVAL_HPP

#include "THINWALL_ModelEval0.hpp"

class THINWALL_ModelEval0_w_Resid : public THINWALL_ModelEval0
{
  public:

    THINWALL_ModelEval0_w_Resid(LIME::Problem_Manager & pm, const string & name);

    virtual ~THINWALL_ModelEval0_w_Resid() {};

    // Methods in addition to those in THINWALL_ModelEval0 that support residual fill callbacks
    virtual EpetraExt::ModelEvaluator::InArgs createInArgs() const;
    virtual EpetraExt::ModelEvaluator::OutArgs createOutArgs() const;
    virtual void evalModel(const InArgs&, const OutArgs&) const;

    virtual Teuchos::RCP<const Epetra_Vector> get_x_init() const;
    virtual Teuchos::RCP<Epetra_Vector> get_x_state();
    virtual Teuchos::RCP<Epetra_Vector> get_x_state() const;

    void initializeSolution();

    // This should be removed in favor of querying via createInArgs
    virtual bool supports_residual() const { return true; }


  protected:

    // A data layour map needed to create/configure solver objects
    Teuchos::RCP<Epetra_Map> epetra_map_;

    // The ME state vector conformal to the data layout map and that is
    // also used to create/configure solver objects
    mutable Teuchos::RCP<Epetra_Vector> me_interface_soln_;

    // needed to get residual array values
    float* r;
};
#endif
```

D.5 Example 5: JFNK based coupling of ss_con1d and ss_thinwall

This example problem uses the same Multiphysics Driver used in Example Problem 4. In addition, one additional interface routine must be implemented in each of the Model Evaluators. This is an implementation of the "set_x_state" routine. To save space, we only list the additions to the Model Evaluator source (i.e. *.cpp) and header (i.e. *.hpp) files that

are added to the files used for Example Problem 4 to accomplish this requirement.

D.5.1 Additions to the Example 4 CON1D_ModelEval0_w_Resid files needed for Example 5

Additions to the C++ source file

```
.
.
 // ------------CON1D_w_Resid set_x_state ------------------------------------------------

 void
 CON1D_ModelEval0_w_Resid::set_x_state(Teuchos::RCP<Epetra_Vector> x)
 {
   // Copy incoming state into our wrapped application
   for( int i = 0; i < x->MyLength(); ++i )
     (this->t)[i] = (*x)[i];
 }

.
.
```

Additions to the C++ header file

```
.
.
   virtual void set_x_state(Teuchos::RCP<Epetra_Vector>);
.
.
```

D.5.2 Additions to the Example 4 THINWALL_ModelEval0_w_Resid files needed for Example 5

Additions to the C++ source file

```
.
.
 // ------------THINWALL_ModelEval0_w_Resid set_x_state ----------------------------------------------

 void
 THINWALL_ModelEval0_w_Resid::set_x_state(Teuchos::RCP<Epetra_Vector> x)
 {
   // Copy incoming state into our wrapped application
   for( int i = 0; i < x->MyLength(); ++i )
     (this->t)[i] = (*x)[i];
 }
.
.
```

Additions to the C++ header file

```
.
.
   virtual void set_x_state(Teuchos::RCP<Epetra_Vector>);
.
.
```

D.6 Example 6: JFNK based coupling of ss_con1d, ss_neutron and ss_thinwall

D.6.1 Multiphysics Driver for Example 6

```
// ************************************************************************************************
//
// lime_mpd_all_jfnk.cpp
// This is a LIME driver for JFNK coupling the "super-simple" codes ss_con1d, ss_neutron0, and
// ss_ thinwall  using the following Model Evaluators:
//            CON1D_ModelEval0_w_Resid, and THINWALL_ModelEval0_w_Resid
//            CON1D_ModelEval0,         and THINWALL_ModelEval0
//            NEUTRON_ModelEval0
//
// ************************************************************************************************
//
// "#include" directives for the various header files that are needed
//
// c++
#include <exception>
#include <iostream>
#include <string>

// con1d physics header file
#include "CON1D_ModelEval0_w_Resid.hpp"

// neutron physics header file
#include "NEUTRON_ModelEval0.hpp"

// thinwall physics header file
#include "THINWALL_ModelEval0_w_Resid.hpp"

// LIME headers
#include <LIME_Problem_Manager.hpp>

// Trilinos Objects
#include <Epetra_SerialComm.h>

//
// define symbols to be used without qualifying prefix
//
using std::cout;
using std::endl;
using std::exception;
using std::string;

// ------------------------------------------------------------------------------------------------
int main(int argc, char *argv[]) {

  int rc = 0;  // return code will be set to a non zero value on errors
```

```
try {    // the "try" construct is used to allow for catching "exceptions" that might
         // unexpectedly occur.  They can occur at any point in the program's
         // call stack so our exception handling policy is to catch all thrown excptions
         // here in main, output a diagnostic message, and terminate the program cleanly.


  // 1  Create a pointer to an instance of an "Epetra_SerialComm" object, called "comm",
  //    which is a specfic type of "Epetra_Comm" object.  (on the heap).  Note that Trilinos
  //    must be constructed with either an MPI communicator or, if we are not using MPI, a
  //    class that has the same interface as the MPI communicator but does nothing.

        Epetra_Comm* comm = new Epetra_SerialComm;

  // 2  Create an instance of a "Problem_Manager" object called "pm"  (on the stack)
  //    Note: set verbosity on/off by defining the boolean "verbose" as true or false

        bool verbose = false;
        LIME::Problem_Manager pm(*comm, verbose);


  // 3  Create pointers to
  //    an instance of a "CON1D_ModelEval0_w_Resid"    object called "con1d", and
  //    an instance of a "NEUTRON_ModelEval0"          object called "neutron", and
  //    an instance of a "THINWALL_ModelEval0_w_Resid" object called "thinwall"
  //    (on the heap)
  //    Note: There are some Trilinos-specific implementation details here. For example,
  //          Teuchos is a Trilinos library and RCP is a reference counted pointer

        Teuchos::RCP<CON1D_ModelEval0_w_Resid> con1d =
                 Teuchos::rcp(new CON1D_ModelEval0_w_Resid(pm,"CON1DModelEval0_w_Resid"));

        Teuchos::RCP<NEUTRON_ModelEval0> neutron =
                 Teuchos::rcp(new NEUTRON_ModelEval0(pm,"NEUTRONModelEval0"));

        Teuchos::RCP<THINWALL_ModelEval0_w_Resid> thinwall =
                 Teuchos::rcp(new THINWALL_ModelEval0_w_Resid(pm,"THINWALLModelEval0_w_Resid"));

  // 4  Register or "add" our physics problems "con1d", "neutron", and "thinwall" with
  //    the Problem manager. When doing this, we get back values for the
  //    identifier "con1d_id", "neutron_id" and "thinwall_id" from the Problem Manager.

        pm.add_problem(con1d);
        pm.add_problem(neutron);
        pm.add_problem(thinwall);

  // 5  Create and setup the four LIME data transfer operators that we will need

        LIME::Data_Transfer_Operator* p_n2c = new neutronics_2_conduction(neutron, con1d);
        LIME::Data_Transfer_Operator* p_c2n = new conduction_2_neutronics(con1d, neutron);
        LIME::Data_Transfer_Operator* p_t2c = new thinwall_2_conduction(thinwall, con1d);
        LIME::Data_Transfer_Operator* p_c2t = new conduction_2_thinwall(con1d, thinwall);

        Teuchos::RCP<LIME::Data_Transfer_Operator> n2c_op = Teuchos::rcp(p_n2c);
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2n_op = Teuchos::rcp(p_c2n);
        Teuchos::RCP<LIME::Data_Transfer_Operator> t2c_op = Teuchos::rcp(p_t2c);
        Teuchos::RCP<LIME::Data_Transfer_Operator> c2t_op = Teuchos::rcp(p_c2t);


  // 6  Register or "add" our four transfer operators with
  //    the Problem manager

        pm.add_preelimination_transfer(c2n_op);
        pm.add_preelimination_transfer(c2t_op);
        pm.add_postelimination_transfer(n2c_op);
        pm.add_postelimination_transfer(t2c_op);

        pm.register_complete(); // Trigger setup of groups, solvers, etc.
```

```
    // 7  We are now ready to let the problem manager drive the coupled problem

            pm.integrate();

  }
  catch (exception& e)
  {
    cout << e.what() << endl
         << "\n"
         << "Test FAILED!" << endl;
         rc = -1;
  }
  catch (...) {
    cout << "Error: caught unknown exception, exiting." << endl
         << "\n"
         << "Test FAILED!" << endl;
             rc = -2;
  }

  return rc;
}
```

DISTRIBUTION:

1	MS 1426	S. Plimpton, 1426
1	MS 1320	W. Spotz, 1442
1	MS 1318	R. Hooper, 1445
1	MS 1318	R. Pawlowski, 1444
1	MS 1323	R. Schmidt, 1444
1	MS 1321	R. Summers, 1444
1	MS 0836	A. Lorber, 1541
1	MS 0748	N. Belcourt, 6232
1	MS 0899	RIM-Reports Management, 9532 (electronic)

Sandia National Laboratories